Computer Science Coursework - Bughouse

Daniel Groves

March 2023

Contents

1 Analysis **3**
- 1.1 The Problem . . . 3
- 1.2 My Stakeholder . . . 4
- 1.3 Existing Solutions . . . 5
  - 1.3.1 Chess.com . . . 5
  - 1.3.2 Lichess.com . . . 8
  - 1.3.3 Chess24.com . . . 9
  - 1.3.4 Case study conclusion . . . 10
  - 1.3.5 Solution Structure . . . 10
  - 1.3.6 Piece Representation . . . 11
  - 1.3.7 Board Representation . . . 11
  - 1.3.8 Check Detection . . . 12
  - 1.3.9 Checkmate Detection . . . 12
- 1.4 Essential Features . . . 13
- 1.5 Limitations of the Proposed Solution . . . 13
- 1.6 Solution Requirements . . . 14
- 1.7 Measurable Success Criteria . . . 14

2 Design **15**
- 2.1 Structure . . . 15
- 2.2 Key Variables . . . 16
  - 2.2.1 Class . . . 17
  - 2.2.2 Globals . . . 17
  - 2.2.3 Functions . . . 17
- 2.3 Usability Features . . . 17
- 2.4 Validation . . . 19
- 2.5 Designing standard Chess . . . 19
- 2.6 Designing the interaction between boards . . . 32
  - 2.6.1 The Server . . . 33
  - 2.6.2 The Clients . . . 34
  - 2.6.3 The Server Part 2 . . . 36
  - 2.6.4 The Client Part 2 . . . 38
- 2.7 Test Plan . . . 41
  - 2.7.1 Testing Knight Moves . . . 41
  - 2.7.2 Testing Queen Moves . . . 43
  - 2.7.3 Testing Checkmate . . . 45

3 Developing the coded solution **50**
- 3.1 Development Cycle 1 . . . 50
  - 3.1.1 Aims . . . 50
  - 3.1.2 Development . . . 50
  - 3.1.3 Prototype(s) . . . 82
  - 3.1.4 Review . . . 82

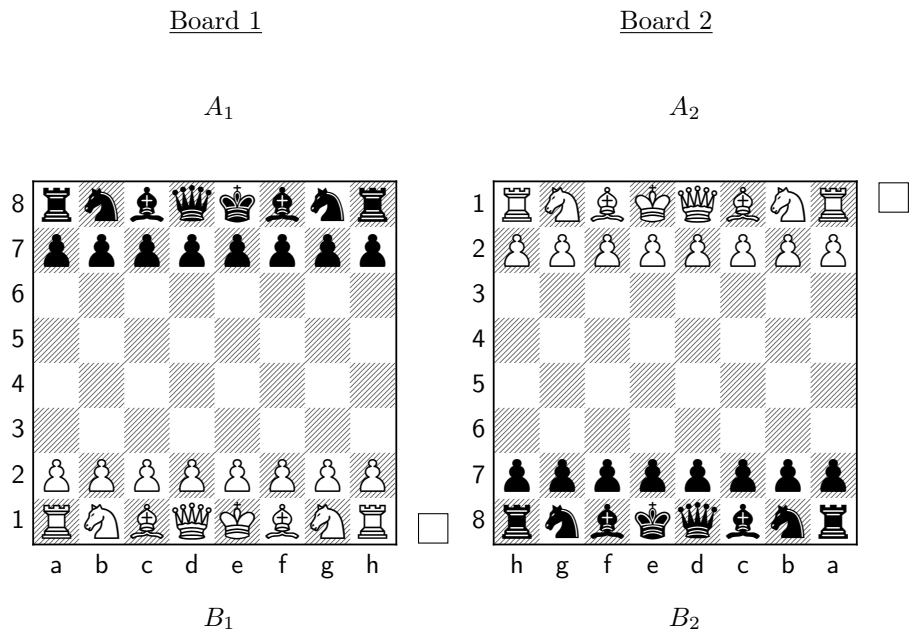
3.2 Development Cycle 2 83
3.2.1 Aims 83
3.2.2 Development 83
3.2.3 Prototype(s) 99
3.2.4 Review 100
3.3 Development Cycle 3 101
3.3.1 Aims 101
3.3.2 Development 101
3.3.3 Prototype(s) 103
3.3.4 Review 103

4 Evaluation **103**
4.1 Testing to Inform Evaluation 103
4.1.1 Post development testing 103
4.1.2 Usability testing 105
4.1.3 Stakeholder Interview 105
4.2 Success of the solution 106
4.3 The Final Product 108
4.4 Maintenance and Further Development 109
4.4.1 Maintenance 109
4.4.2 Further Development 110

1 Analysis

1.1 The Problem

"Bughouse" is a variant of Chess played by four players, with two teams of two. There are two separate boards. Take the two players on Team A to be A_1 and A_2, and the two players on Team B to be B_1 and B_2. The board set-up would look as so:



In Bughouse, a player's captured piece can be placed on their teammate's board. The game is usually fast-paced with a blitz time control, requiring quick thinking and adaptation.

Due to the nature of a fast paced game, capturing, placing and passing pieces to each other becomes chaotic quite quickly, giving warrant to its name: "Bughouse", which is slang for a mental hospital. It is hard to solve this problem of increasing chaos over the board. Although a slower time control would decrease the chaos, it would also remove some of the charm and entertainment of the game.

It seems to me that using computational methods would allow this problem to be solved. The game could be played at a fast pace whilst avoiding the chaos. It means pieces can be captured, placed and "passed" seamlessly without being

knocked over et cetera. It would also mean that players would not have to worry about obeying the rules in time pressure as the computer would ensure they cannot make illegal moves, a feature that I think would be highly desired by beginners. As it stands, it seems like computational methods are the only way to solve the problem of chaos when playing quickly, reinforcing its suitability as a solution to this problem.

As well as this, Bughouse is a game that allows friends to play chess on a team, as opposed to against each other like in conventional chess. This reason reinforces my belief that the game would be desirable to play for beginners who would have some support from their teammate.

1.2 My Stakeholder

As such, it seems like a reasonable stakeholder would be a beginner at chess who can use my creation of Bughouse on the computer to play with a teammate against other players. It would be much less pressurising for them, and would also help them learn the basic rules of chess whilst they play, which is exactly what a beginner needs. For that reason I have chosen my friend Zack to be my primary stakeholder. He is a beginner at chess who wants to get better and improve his understanding of the game whilst having some guidance from a teammate. Below is my interview with him, to get his thoughts and opinions on things before I begin my analysis.

Daniel: Hi Zack. I'm currently in the process of developing a software to help beginner chess players improve their skill. What do you think about that? What do you think would be useful for a beginner chess player like yourself?

Zack: As a beginner, it is important to me that I have guidance on making legal moves as I am not that familiar with the rules and would like to be more confident on them. The main issue I have as a beginner player , as well as not being familiar with legal moves, is making good moves to help win the game.

Daniel: Thank you. Those both sound like things that most beginners would want help with. My plan is for the software to only allow you to make legal moves. That way, you would learn the moves and not have to worry about making illegal moves.

Zack: That sounds great! I think it would be much less stressful to be guided through making legal moves

Daniel: I am also planning to create a chess variant that is a 2v2 format. Although it requires all the same skills it would allow you to be guided by a more experienced teammate. That way you would not only get guidance on making legal moves but also on making "good" moves.

Zack: Playing with a teammate sounds much more fun as well!

1.3 Existing Solutions

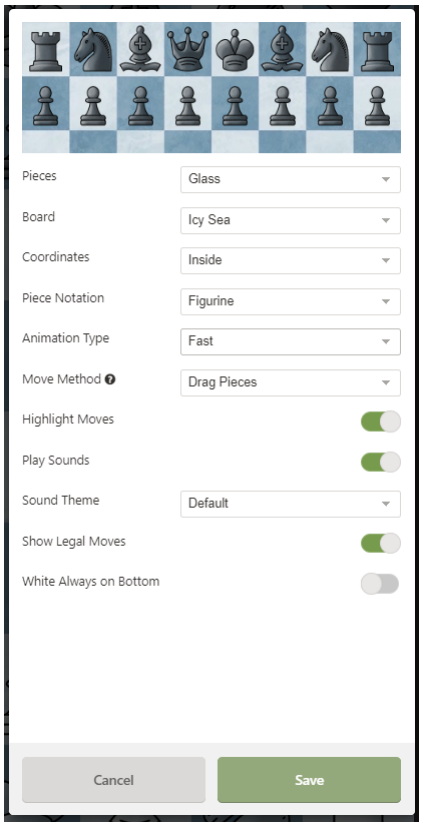
There are numerous well developed websites that allow users to play chess online, either with friends or online opponents such as **chess.com**, **lichess.org** and **chess24.com**. However, at the moment only chess.com, the most popular website, has an implementation of bughouse. As the website has been around for 15 years, the chess interface is very smooth in my opinion. As bughouse can use a lot of the same code that regular chess can it is also rather smooth, although it feels to me as though certain unique bughouse features like placing pieces do feel more clunky.

Personally, I think chess.com shows that one suitable approach of tackling the problem is to program a regular chess interface first and then implement the bughouse features. Unfortunately, it is not public knowledge exactly what programming languages chess.com uses but it is suspected that PHP is used for the website and Java is used on the backend.

1.3.1 Chess.com

By far the biggest chess website, chess.com, with almost 100 million users is the only website I could find that lets users play bughouse. However, I will first investigate the standard chess. As mentioned above I will mainly focus on the aspects clear to the user as I cannot get much insight into how the game itself is coded.

The chess.com GUI is rather advanced. One feature of note that may be worth considering are the highlights which show the most recent move made to make it clear to players what has happened if they look away for a moment. Chess.com also allows the user to use their arrow keys or click the two arrow buttons in the bottom right to navigate between positions have happened in the game. Although this feature seems extremely useful it may beyond the scope of my project, it would not only complicate the GUI but also require storing all positions in the game rather than just the current one. That could involve storing over 100 8 by 8 arrays.



Chess.com also allows the user to customise the piece and board style as well as the animation type amongst other things. Once again, although this is a nice feature that I could look into in the future it seems out of the scope of my project.

Finally, I will look at the implementation of bughouse in chess.com:

Chess.com splits the GUI into two sides. The left hand side which has a width of about 2 thirds of the page contains the main board of the user and shows the time for both the user and the opponent. On the right hand side, taking up the remaining third, a smaller board which shows the user's teammate's board is displayed along with their clock. There is also a space to chat or send emotes to teammates to discuss strategy.

I rather like this design and will consider it when designing my own GUI. I expect my own implementation will contain the essential features like the two boards and clocks but may not include the additional features like a live chat.

Although I cannot say for certain I assume chess.com processes all the moves on a server and updates the users' boards accordingly.

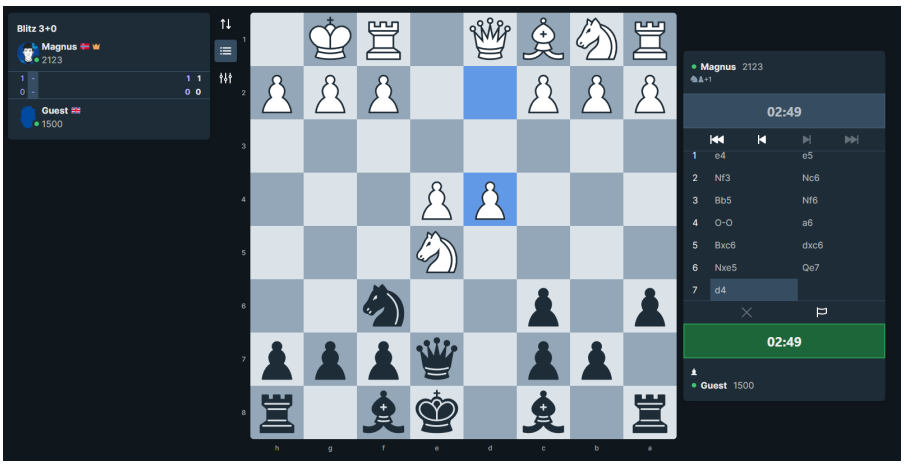
1.3.2 Lichess.com

Lichess.com's standard chess has similar features to that of chess.com but a slightly different layout:

There is not much to comment on lichess.com in contrast to chess.com. It has a different default board style that cannot be customised however it does have settings to customise what is displayed and what is not such as board highlights and piece destinations. To me, lichess.com feels more minimalist than chess.com which may be a positive to some users.

1.3.3 Chess24.com

Once again, chess24.com exhibits similarities to chess.com and lichess.com although with slightly less customisation options it only lets uses choose between a light and dark theme.



1.3.4 Case study conclusion

From my investigation into the 3 most prominent chess websites I have a number of takeaways. Most implementations of standard chess are rather similar, as you might expect. The main 2 features the 3 websites I investigated have that I didn't consider are:

- Showing the history of moves
- Allowing the user to move between past positions

Although I don't think these features are particularly necessary, they are definitely ones to keep in mind if I have the time to perfect the GUI of my project.

In terms of bughouse, my main takeaway is some inspiration as to how I could layout the two boards, with one larger and one smaller. However, as I could only find one website that has bughouse, there is definitely lots of room to experiment and find a different layout and it is probably worth considering whether it is necessary to display both boards on one screen.

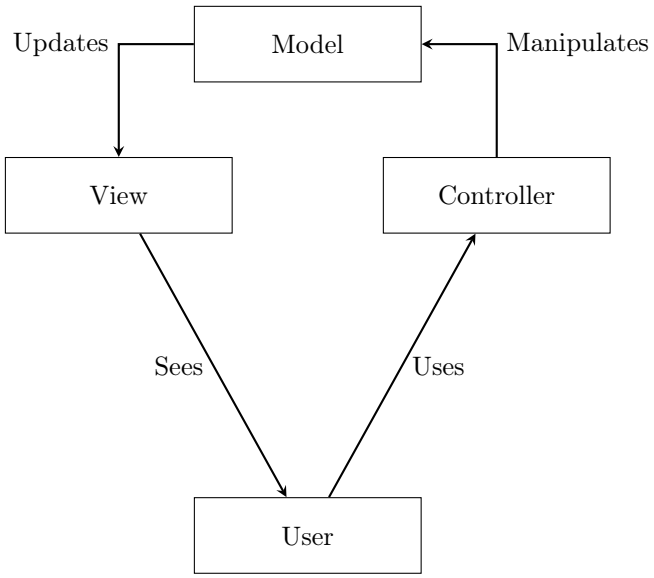
As well as this, I have more in mind a game where teammates are in the same room and so they can discuss each other's positions, so having both boards on every screen may not be necessary.

1.3.5 Solution Structure

In terms of solutions to implementing standard chess on a computer there are numerous. Most chess programs and their UIs can be interpreted as a Model-view-controller (MVC). An MVC is a software architectural structure that divides the program logic into three interconnected elements. The three components are as follows:

- **Model** - The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application
- **View** - Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- **Controller** - Accepts input and converts it to commands for the model or view

In addition to dividing the application into these components, the MVC defines the interactions between them. These interactions can be seen below.



Personally, I think this structure suits the game of chess perfectly. It seems logical that a computational solution could be split up into model, view and controller with minimal overlap so this seems like a suitable approach to think about when designing my solution.

1.3.6 Piece Representation

There are three main approaches to displaying pieces in existing solutions.

The first approach is to use 2D vector graphics to draw the pieces on the board. For each piece an array of connected lines is declared, the first one a close polygon, which is filled by the piece color.

The second approach is to use Bitmaps: small, pre-painted images stored as raster graphics in an external file.

The final approach for displaying pieces is by using symbols in Unicode.

Although I can see the appeal of the first approach when making a tidy solution, it seems unnecessarily complicated to me. The last two however seem like justified approaches for displaying the pieces without over complication.

1.3.7 Board Representation

There are two main techniques to board representation in existing solutions.

The first technique is known as **Piece Centric**. A piece centric representation keeps lists, arrays or sets of all the pieces still on the board, with the associated information of which square they occupy.

The second technique is known as **Square Centric**. A square centric representation has a list or array of all the squares on the board and its associated information such as if it is occupied by a piece and if so, which piece.

From my perspective, both techniques seems suitable for the problem. However, I think a piece centric technique would be preferable to me as it is very easy to visualise how the phyiscal pieces are being stored, and having their information updated as they move around the board.

1.3.8 Check Detection

There are two main strategies to detecting check.

The first strategy involves considering the previous move made by the opponent. The check may be a **direct check**[1] or a **discovered check**[2]. In order to detect both of these one has to test whether the piece on the **target square**[3] attacks the king, or whether the piece on the **origin square**[4] is "attacked" by a piece of its own colour on an otherwise open **ray**[5] in the direction of the opponents king.

The other strategy is slightly different. Rather than directly detecting check it instead tests each opponents move to see whether that is legal, including when in check beforehand. This can be done by testing all the opponents possible moves after a player tries to make a move and sees if any of those moves can capture the king. If they can, we know they were in check beforehand.

The latter strategy kills two birds with one stone in a sense. It recognises check and also sees which moves are legal for getting out of check. However, it involves lots of testing so the first strategy is by far the cheapest way.

1.3.9 Checkmate Detection

Many programs rely on **Pseudo-legal**[6] **move generation**, and detect checkmate if all the generated moves are found as illegal.

There are two main forms of checkmate which have different properties and can be recognised be programs.

There is only one legal form of move when a player is in **double check**[7]:

- The king moves or captures

[1] When a piece is moved into a position where it now attacks the king

[2] When a piece is moved which now leaves a different piece attacking the king

[3] The square occupied by the moving piece after making the move

[4] The square occupied by a piece about to move

[5] Part of a line from a starting square in one particular direction, the length of a ray can vary from 0 to 7 squares

[6] A move that can be made according to the moveset of the piece being moved, but not necessarily legal due to check/checkmate

[7] When multiple pieces are checking the opponents king

There is some more freedom in forms of move when a player is only in **single check**[8]:

- Capture the checking piece
- The king moves or captures
- Placing an interposing piece between the sliding piece and king

1.4 Essential Features

From my research I have come up with a list of essential features for my proposed computational solution:

- Each player must be able to see the board they are playing on and all the moves made, as well as when and how checkmate is delivered
- When a player takes a piece it must appear in an area that shows their teammate can place it (and vice versa). Again this is an important feature to allow the game to work.
- A player must be able to drag a piece in the area mentioned above onto the board as one of their own pieces
- Each board must have a clock to keep the game fast paced as originally wanted. If a single player runs out on time, their team has lost. Although a clock isn't vital to the game, it is what makes the game stay entertaining and one of the reasons for the computational solution to begin with.
- Each player should hear sounds when moves on their board are made

1.5 Limitations of the Proposed Solution

There are some limitations to my proposed solutions, which mainly centre around the interactions in the MVC structure and keeping information consistent across all players.

- Where do the calculations take place in a way that allows all the board information to be sent to each player in realtime?
- How should clock times be kept in sync?

The other limitations of my proposed solution mainly arise in places where Bughouse differs from standard chess.

- A clock running out on any board will end the game for all players
- A win or draw on any board must end the game for all players
- Pawns may not be placed on the 1st or 8th ranks

[8] When only one piece is checking the opponents king, this is much more common than a double check

As well as this there are certain miscellaneous limitations that cannot be obviously grouped but may be significant

- As discussed above, most existing programs detect checkmate by finding a situation where no pseudo-legal moves are legal in the current check. Testing all of these may cause some delay which must be minimised to keep the game smooth.
- Four players are required to play the game and it may be difficult to find 4 players of a similar standard who want to play. One way of helping with that is matching the weakest player of each team with each other and the strongest with each other.
- The correct code must be distributed to the players alongside an explanation of how to run the code

1.6 Solution Requirements

- A server will need to make all the calculations and complete any processing.
- The server will receive inputs and send outputs to and from players
- Each player must act as a separate client to the server

Below is a table that summarises the details of what software will be required:

Program/Module	Version	Who needs it	Justification
Python	3.8	Server and Players	Main Programming language
Pygame	2.1.2	Players	To display the board
Numpy	1.23.5	Server	For easy calculation to return sign of number
Pytest	7.2.0	Server	To test functions
Time	Built-in Module	Server and Players	To control the refreshing of the board
Socket	Built-in Module	Server and Players	To communicate between clients and servers
Pickle	Built-in Module	Server and Players	To serialize and deserialize data

1.7 Measurable Success Criteria

Unfortunately, as I am making a game it is hard to have objective and measurable success criteria. As a result of this I shall measure my success through the thoughts of my stakeholder. Throughout various stages of the process I shall ask my stakeholder to rate various aspects from a scale of 1-10 and use this to measure my success. The aspects will be as follows:

- Intuition to use
- Smoothness of gameplay
- Sizing of the UI
- Aesthetics of the UI
- Fulfillment of essential features

Earlier I included an **interview** with my stakeholder, Zack. I will conduct interviews with him at various stages of the process to gauge success and get direct feedback.

As well as using my stakeholder to measure the success criteria there are also a number of objective criteria that I will use to evaluate the success of my project throughout:

- The player should be able to drag and drop pieces to different squares
- The program should only allow legal moves to be made
- Pieces should be able to be captured and placed by a teammate
- The program should recognise when a player is in check, and only allow moves that get the player out of check to be made
- The program should recognise when checkmate has been delivered
- The program should allow special moves such as castling, promotion and en passant to be made
- The program should recognise when a draw has been made in the form of stalemate, repetition or the 50-move rule

These success criteria are all important aspects of the game that I have gathered from speaking to my stakeholder, and my own research and knowledge of the game.

2 Design

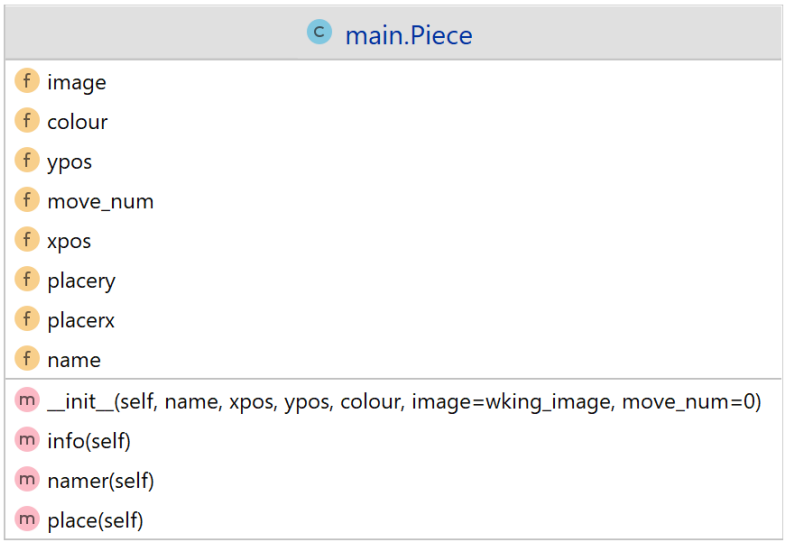
2.1 Structure

From my investigation into existing solutions I think the **MVC Structure** will suit the problem very will. By looking at the problem in this way I can break the problem down into 3 subsections: The model,view and controller. The model will perform all calculations and checks and will update the view of the players. The user can put in inputs(controller) and the model will process these.

This still seems like a rather large problem to tackle at once. I think the best way forward will be to program standard chess. This way I will create the main algorithms required(which I will go into later) that will be needed for Bughouse and yet the testing will be much easier.

Once I have a working solution for standard chess I can then use the majority of it as the main component of the controller. I will also be able to use some of it for the view to display the board.

From my analysis, I think a **Piece Centric** representation will work best as it is very intuitive for the pieces to be objects which have information about which square they occupy. Here is a class diagram for a piece.



Piece Class diagram

The **info** and **namer** methods display the position and name of the piece, respectively. These methods were added for my own convenience while programming to help me understand the state of each object during execution. The **place** method was also created for my own convenience, in order to create a text version of the board to look at.

In the future I may create sub-classes for each individual piece making use of inheritance. I will see how this goes in the development stage and update in a future iteration if needed.

2.2 Key Variables

Below are details of the key variables that will be used in the development of my project.

2.2.1 Class

Class Name	Attribute name	Description
Piece	name	The name of the piece
Piece	xpos	The x-coordinate of the piece (square)
Piece	ypos	The y-coordinate of the piece (square)
Piece	Colour	Colour of piece
Piece	image	The image of the piece
Piece	placerx	The x-coordinate of the piece (pixel)
Piece	placery	The y-coordinate of the piece (pixel)
Piece	move_num	The amount of time the piece has been moved

2.2.2 Globals

Name	Description
ap	A list of all the piece objects
wp	A list of all the white piece objects
bp	A list of all the black piece objects
Move	Whether it is white's turns (True or False)
Checking pieces	A list of all the pieces checking the king

2.2.3 Functions

Function	parameter	Description
piece_moves	x	The x component of attempted move (vector)
piece_moves	y	The y component of attempted move (vector)
piece_moves	startx	The rank a piece began before the attempted move (square)
piece_moves	starty	The file a piece began before the attempted move (square)
pawn_moves	takenpiece	Whether the attempted move is a capture
white_pawn_moves	wpa	The pawn trying to move
black_pawn_moves	bpa	The pawn trying to move
bishop/queen/rook_moves	vectorx	Vector to a given rank along the attempted move
bishop/queen/rook_moves	vectory	Vector to a given file along the attempted move

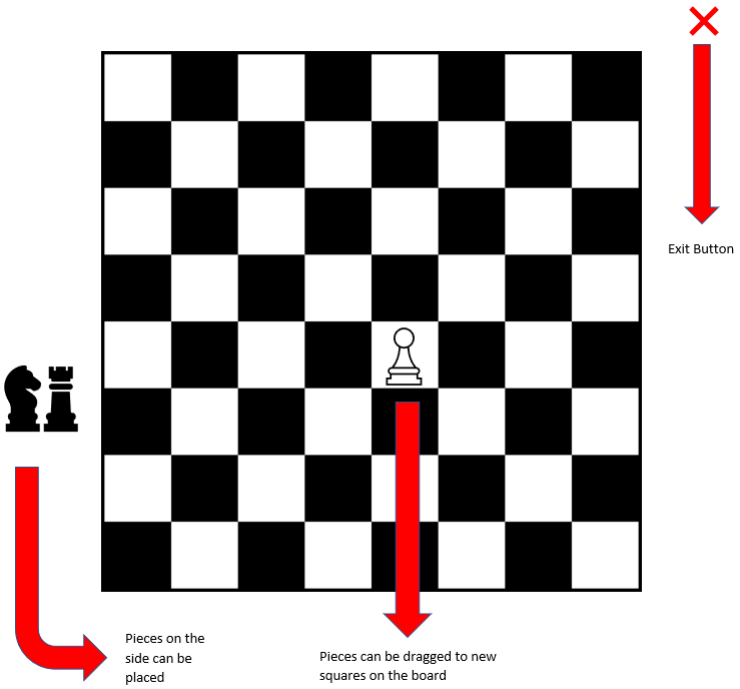
In the table above I use **piece_moves** to refer to all the separate piece move functions. Similarly, **pawn_moves** refers to both the white and black pawn moves functions.

2.3 Usability Features

Below are the usability features to be included in the final solution:

Feature	Justification
Pieces can be dragged and placed on new squares	Essential aspect of chess
Captured pieces can be placed from the side of the board	Essential aspect of bug-house
The window can be shut	Intuitive to the player and reduces clutter
The colours of the pieces and board should be clear	Will make it much easier for beginners or those with vision issues to play
The pieces should be big enough and each type of piece should be clearly recognisable	Will make it much easier for beginners or those with vision issues to play

The majority of the usability features simply involve allowing the player to play chess. In terms of physical features of the GUI this just involves placing/capturing pieces as well as an exit button. There is no need for a fancy exit button, I will just use the regular exit button on a pygame window as it is where a user instinctively goes to exit.

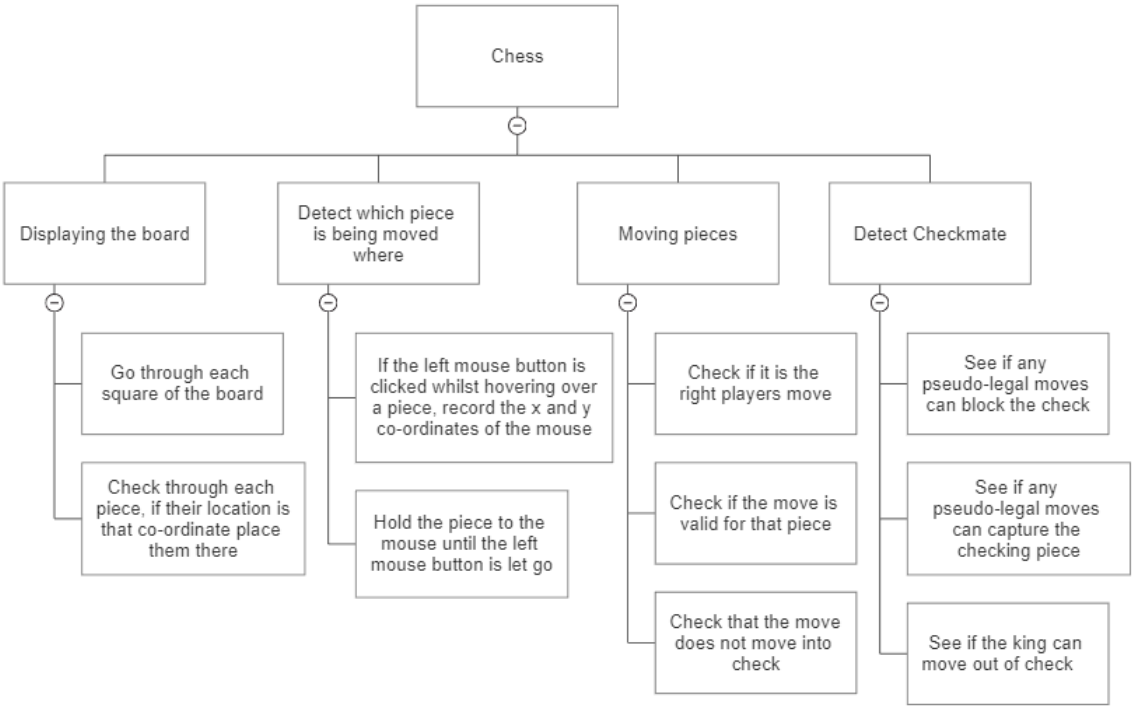


2.4 Validation

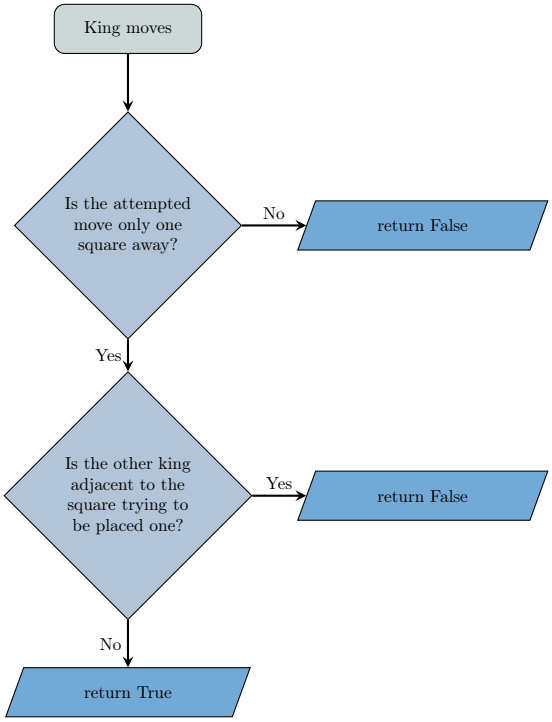
Due to the restriction of physical space, it is very difficult for users to input invalid data as they can only put in physical co-ordinates. The one thing I do need to handle is when a user tries to place a piece off the board. This will be quite easy to take care of in the functions that decide whether a pieces move is valid.

2.5 Designing standard Chess

Below is a breakdown of the logic to the game of chess:

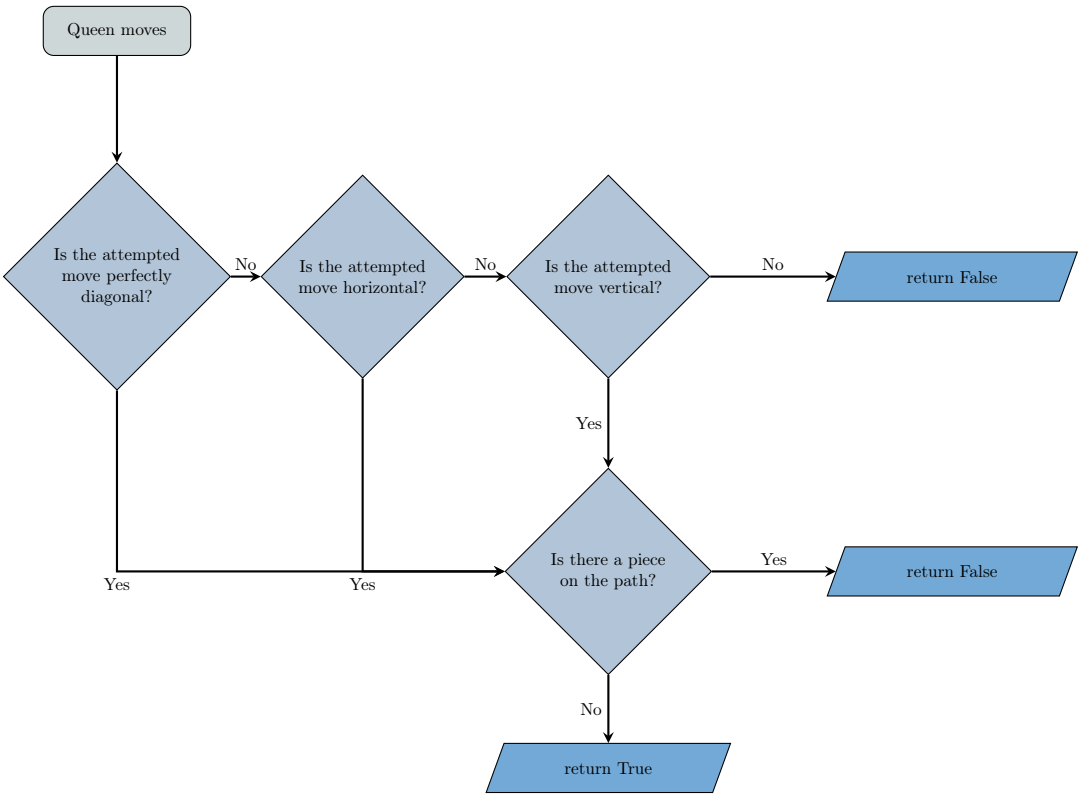


Now I will include flowcharts of the logic for some of the more complex individual boxes in the chart above. Checking if the move is valid is perhaps the most fundamental logic of the code. For each individual piece there will be a function to see if the attempted move is legal, I will show flowcharts and the accompanying pseudocode of these below.



Algorithm 1 King Moves

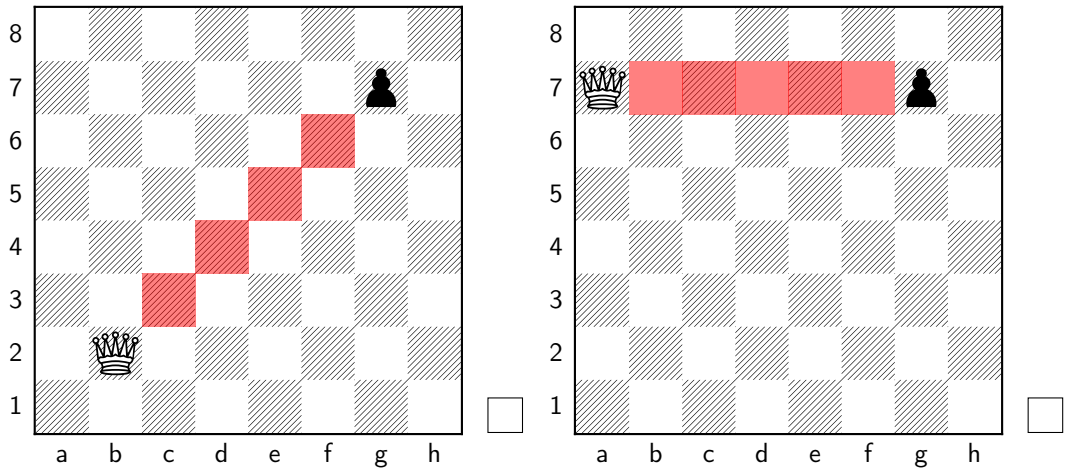
```
procedure KINGMOVES(X, Y, STARTX, STARTY, ITEM)
    if abs(x) < 2 and abs(y) < 2 AND NOT piecethere(startx + x, starty + y, item) then
        SET returner TO True
        for vector IN king_vectors do
            if item.name[0] EQUALS "w" then
                if bking.xpos EQUALS startx + x + vector[0] AND bking.ypos EQUALS starty + y + vector[1] then
                    SET returner TO False
                    break
                end if
            else
                if wking.xpos EQUALS startx + x + vector[0] AND wking.ypos EQUALS starty + y + vector[1] then
                    SET returner TO False
                    break
                end if
            end if
        end for
    else
        SET returner TO False
        return returner
    end if
end procedure
```



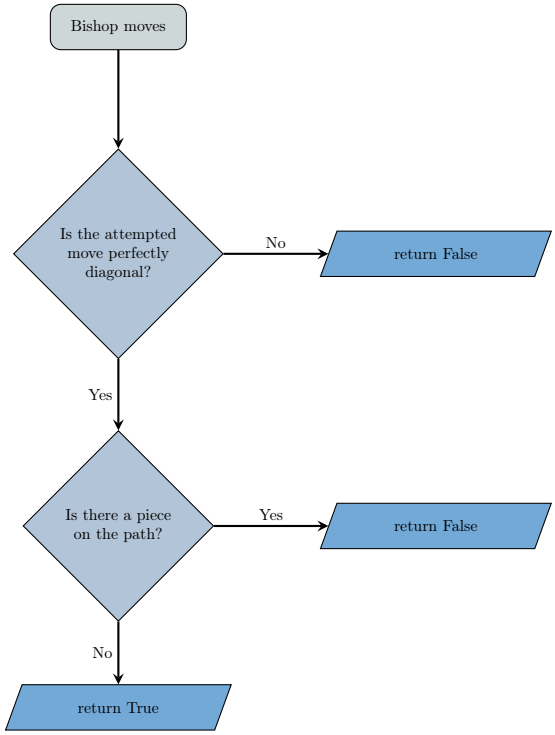
Note that although the above flowchart, shows the first 3 decisions all feeding into 1 if the answer is yes, in reality they will not be the same function in the code as seeing if there is a piece on the path will be different depending on whether the piece is moving vertically, horizontally or diagonally.

As well as this the use of "perfectly" diagonal is to denote that the change in the x co-ordinate of the piece must be the same as the change in the y co-ordinate of the piece. Otherwise a diagonal could include, for example, moving 1 square up and 3 squares to the right.

Below are examples to demonstrate the "Is there a piece on the path?" decision where the queen is trying to move from its current position to capturing the pawn on **g7** and the highlighted squares are those that would need to be checked for other pieces.

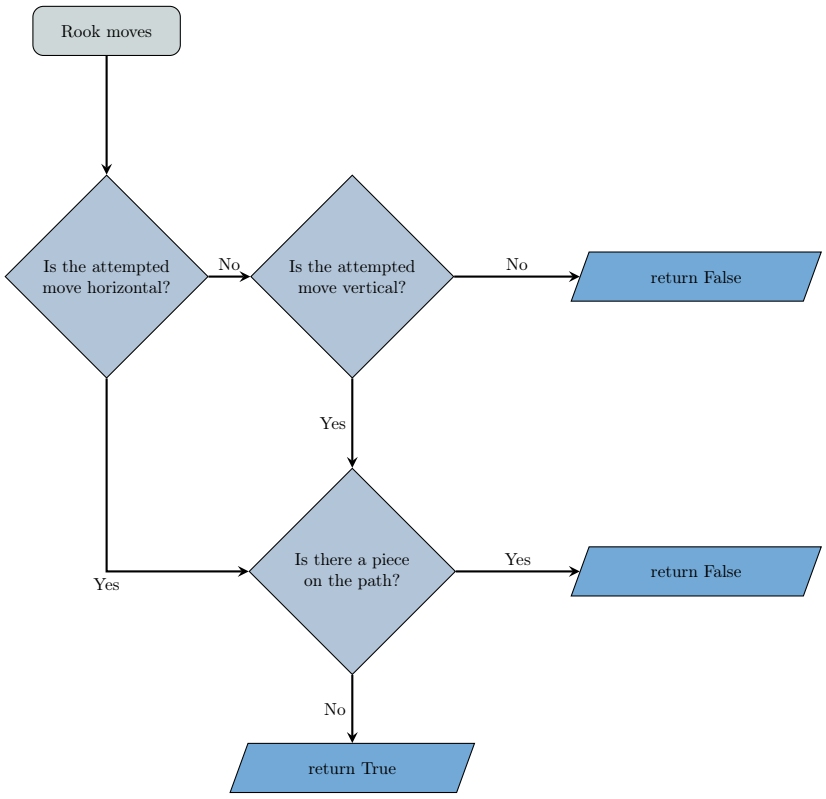


The flowchart for the queen moves is quite helpful for creating ones for the bishop and rook. The moveset of a queen is essentially that of a rook and bishop combined.

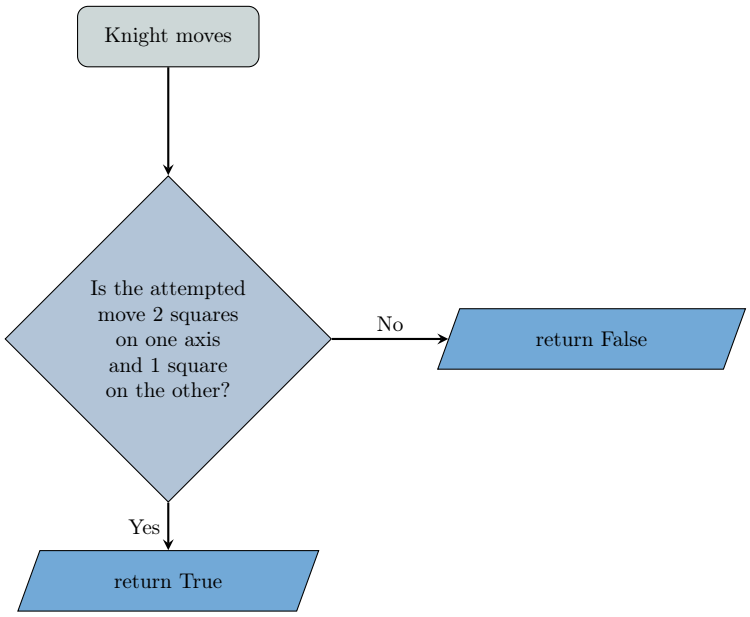


Algorithm 2 Bishop Moves

```
procedure BISHOP_MOVES(X, Y, STARTX, STARTY, BISHOP)
    if abs(x) EQUALS abs(y) then
        for square IN range(1, abs(x) + 1) do
            if x < 0 then
                SET vectorx TO startx − square
            else if x > 0 then
                SET vectorx TO startx + square
            end if
            if y < 0 then
                SET vectorx TO starty − square
            else if y > 0 then
                SET vectory TO starty + square
            end if
            if abs(x) EQUALS squareandpiecethere(vectorx, vectory) then
                return True
            else if piecethere(vectorx, vectory) then
                SET returner TO False
                return returner
            else
                SET returner TO True
            end if
        end for
    else
        SET returner TO False
    end if
    return returner
end procedure
```



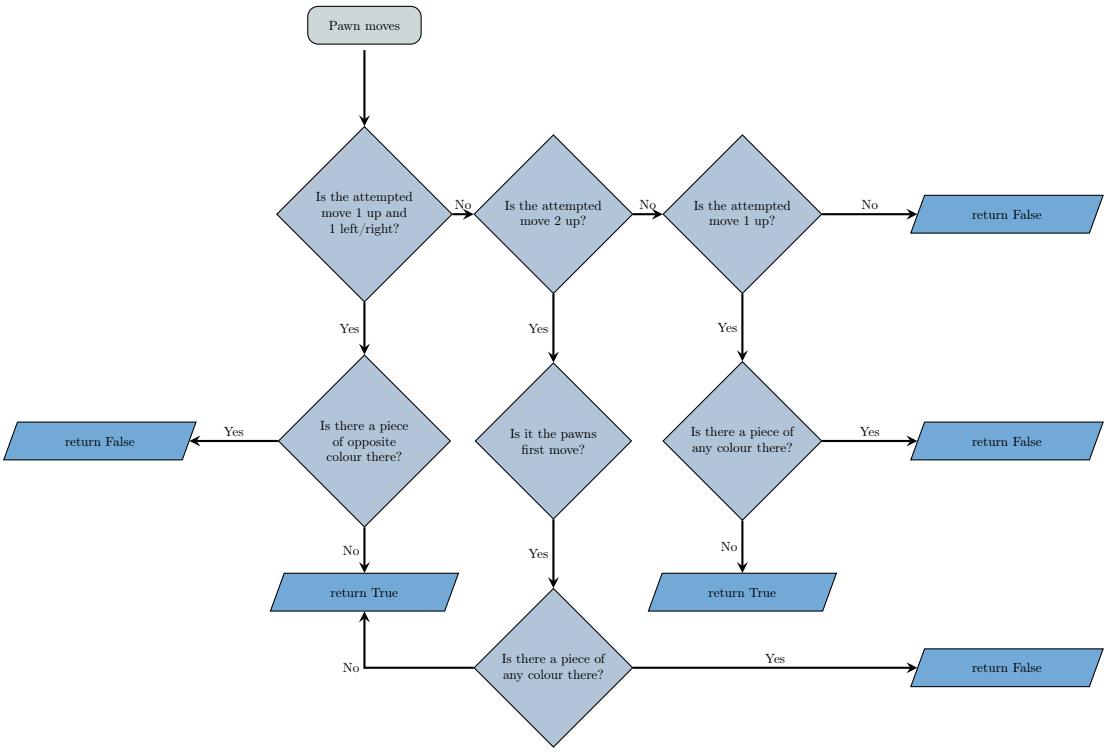
The knight is a rather unique piece on the board in that it can jump over pieces. Because of this it actually make the algorithm to return the validity of an attempted knight move quite simple.



Algorithm 3 King Moves

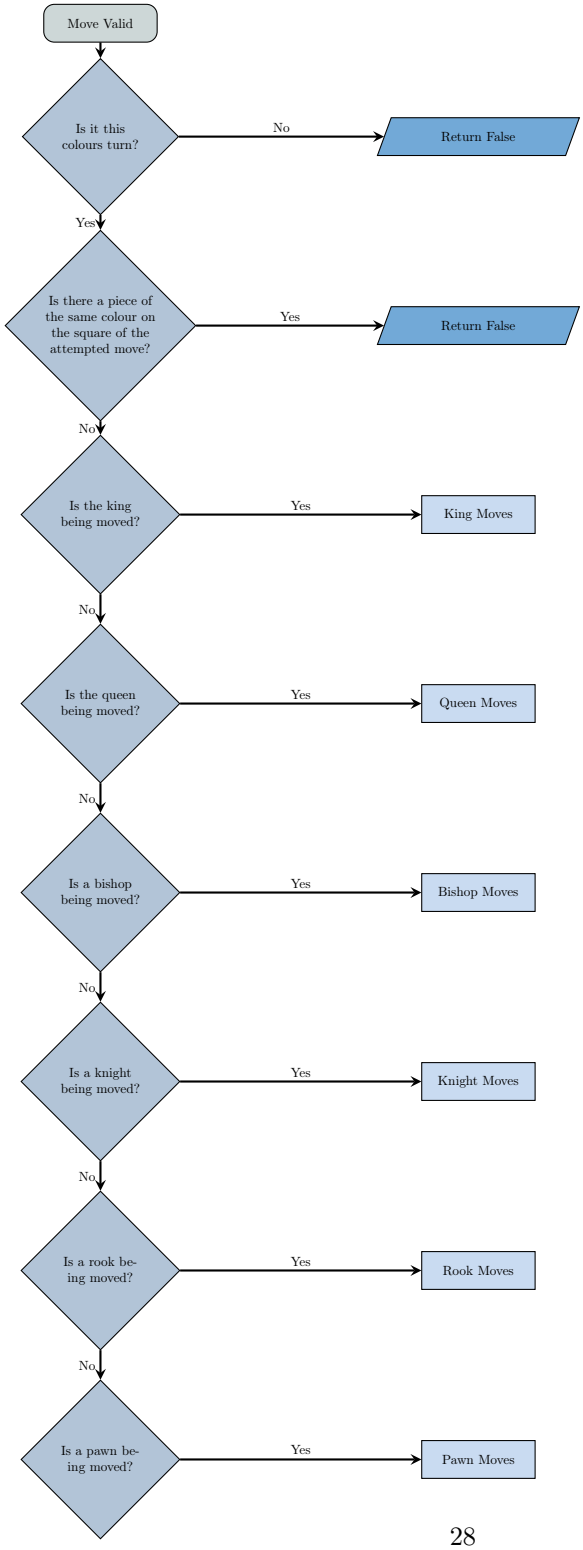
```
procedure KNIGHT_MOVES(X, Y, STARTX, STARTY, ITEM)
    if (abs(x) EQUALS 1 AND abs(y) EQUALS 2) OR (abs(x) EQUALS 2 AND abs(y) EQUALS 1) then
        if piecethereexclude(startx + x, starty + y, knight) OR NOT piecethere(startx + x, starty + y, knight) then
            return True
        end if
    else:
        return False
    end if
end procedure
```

Pawns are also a rather unique piece in that each colour can only move one way up the board. For this reason I will most likely implement an algorithm for each colour. However in the flowchart below I will use "up the board" to mean away from the player moving to generalise for both colours.

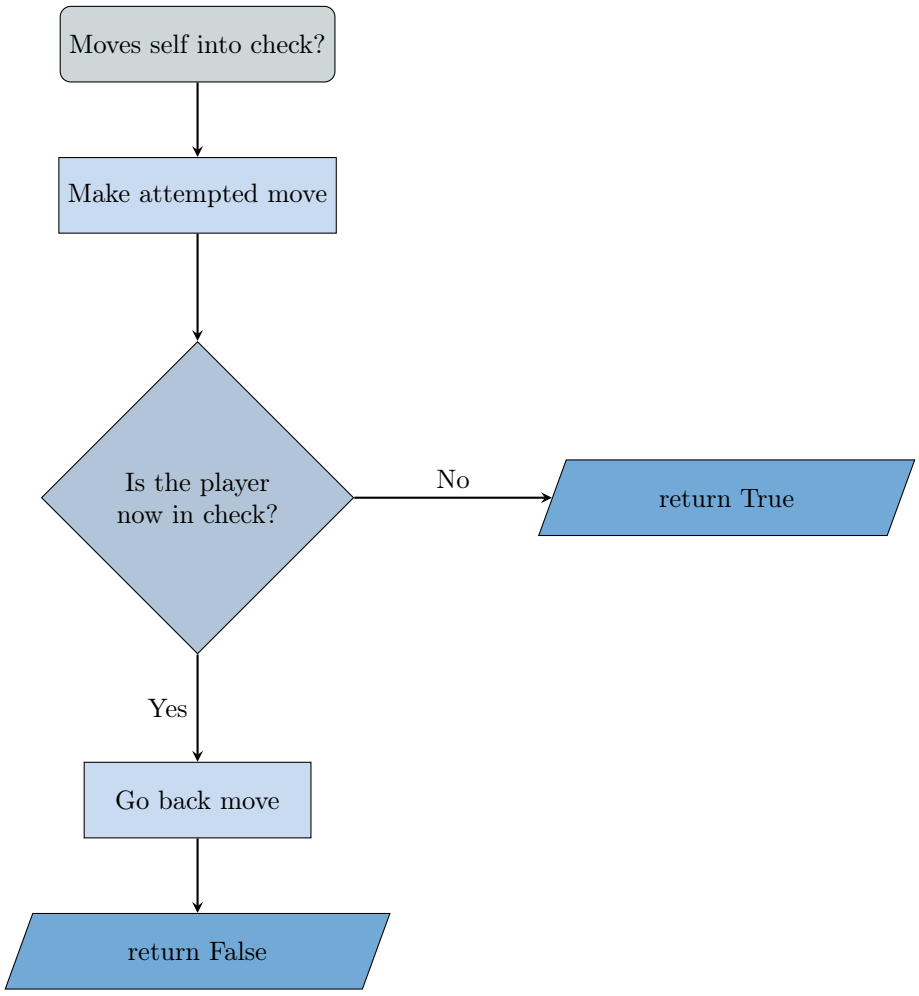


That cover's the algorithms for returning the validity of a move for all piece types. The one thing that is not included in those algorithms is that a piece can never move to a square if a piece of the same colour is there. This will be covered in the main algorithm as an attempt to move to a square with a piece of the same colour will be false no matter what.

The algorithm that decides which function should be used is rather simple:

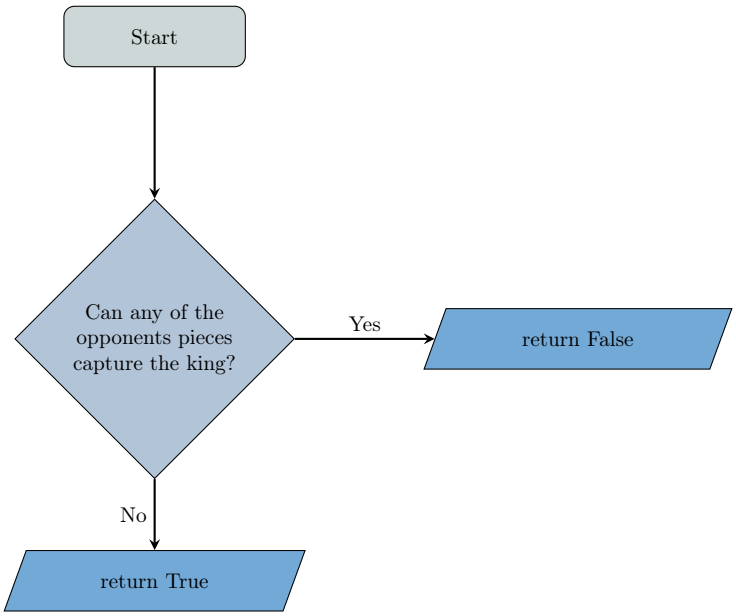

Move Valid
Is it this colours turn?
No
Return False
Yes
Is there a piece of the same colour on the square of the attempted move?
Yes
Return False
No
Is the king being moved?
Yes
King Moves
No
Is the queen being moved?
Yes
Queen Moves
No
Is a bishop being moved?
Yes
Bishop Moves
No
Is a knight being moved?
Yes
Knight Moves
No
Is a rook being moved?
Yes
Rook Moves
No
Is a pawn being moved?
Yes
Pawn Moves

As shown in the **main overview**, once we have checked a move is valid according to the pieces move-set we must also ensure that the move does not put ones self in check. As a check is defined by a board's position, as opposed to a movement, I think the best way to do this is to simulate the move. First we simulate the move, if the position is now check(where the colour that just moved is in check) we know the move was not valid and so we move the board position back to what it was before.



This logic is rather simple, however the decision block, "Is the player now in check?", can be broken down into an algorithm itself. One way of thinking

about check, is it means if the person in check did nothing (against the rules of the game!) the attacker would be able to capture the opponents king on the next move. Thinking about it like this we can use the individual move set functions to return whether it is valid for a piece to capture the king on the next go (again, if nothing were to be done), in which case we know it is check.



In the "Can any of the opponents pieces capture the king?" block, we will need to loop through all the opponents pieces and see if any of them can capture the king using their respective move set functions shown above.

The final part of the **main overview**, and perhaps the most complicated, is **detecting checkmate**. As discussed earlier there are only 3 ways of getting out of check:

- Capture the checking piece
- Block the check
- Move the king

The first stage to detecting checkmate is detecting check, as checkmate is a special form of check. Therefore, before we run the checkmate checker we can run the check checker, if the check checker returns false we need go no further. However, if check is detected we can use the flowchart above, if a piece can

capture the king on the next move, we know that piece is a checking piece, and we can add it to a list of checking pieces. Note that it is a list as it is possible to have two pieces checking a king at once.

Again we can make good use of the functions I have already designed to see if moves are valid. As well as this we can combine the first two methods of getting out of check in one, by seeing if a piece can move anywhere on the path of the checker (including where the checker is). The reason for recording the checking pieces should now be evident, so we can see if it is possible to take or block the check.

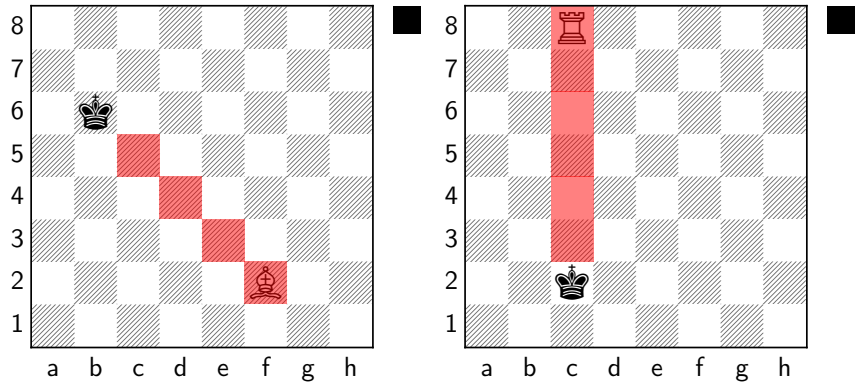
There are five pieces that can put a king in check, however we can already simplify our process by looking at the ways of getting out of check for these specific pieces. All checks have potential for the checking piece to be captured (unless in double check) and for the king to move. However, not all checks can be blocked. The table below summarises which checks can be blocked:

Checking piece	Is it possible for the check be blocked
Queen	Yes
Rook	Yes
Bishop	Yes
Knight	No
Pawn	No

We can see that checks by knights or pawns can never be blocked. This is because knights do not have a ray attack. Therefore a check by a knight cannot be blocked as they target individual squares.

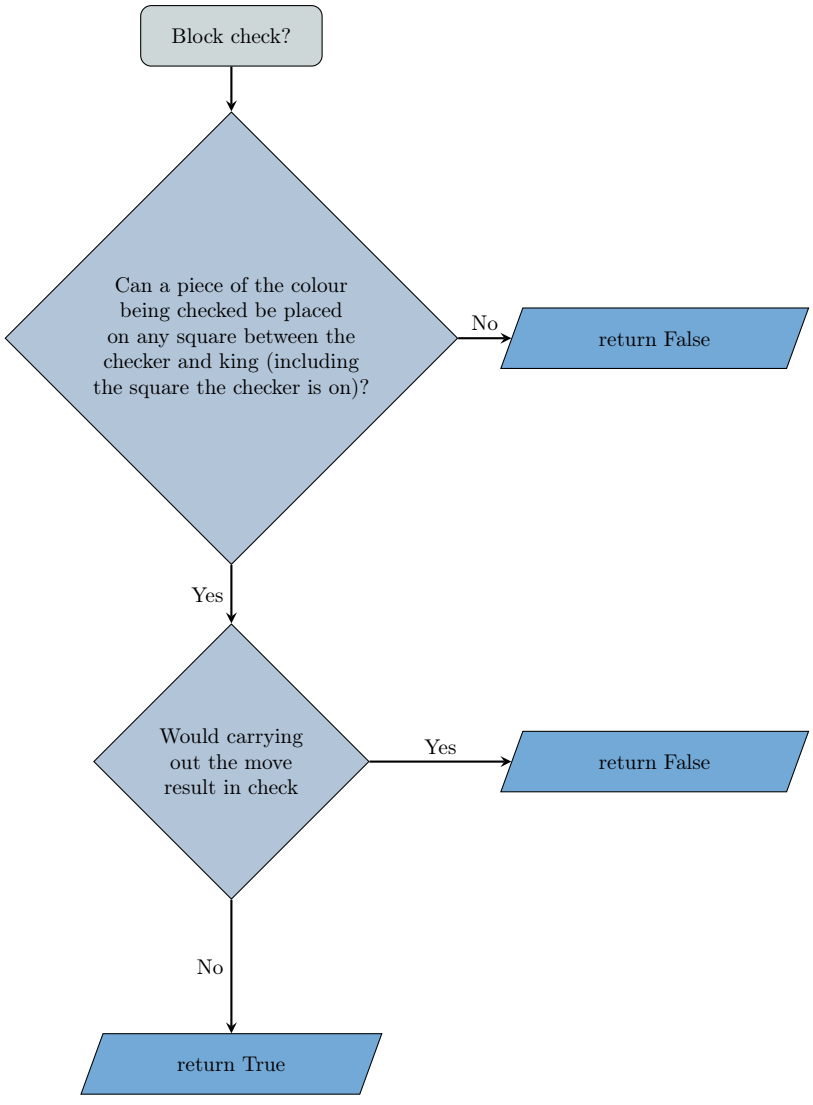
A check by a pawn can also never be blocked as they can only check squares adjacent to them and hence there aren't any squares in between where a piece can be placed to block the check.

Below are examples of the squares we will check if a bishop or rook is the checking piece:



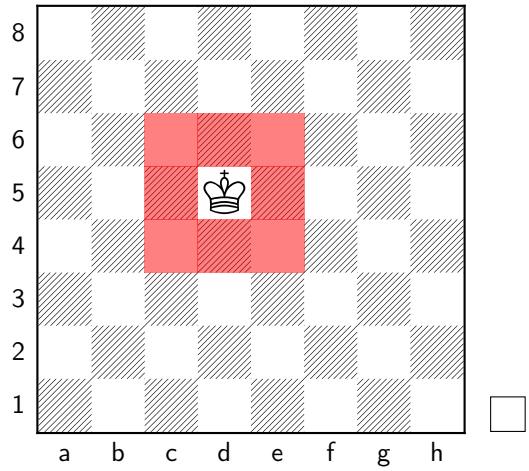
If a queen is the checking piece, we simply have to see if it is performing a diagonal or straight check and then it becomes one of the cases above.

For the complete algorithm, we need to loop through the pieces in our list of checking pieces. Most of the time this will only be one piece, but sometimes it may be two. We will then see what type of piece it is and carry out the respective check.



For the first decision block we simply have to attempt to place a piece on each square, using the above **algorithm**. It still works perfectly as after the attempted move is made, it checks if the position is in check. This will take care of double checks. To find the squares we need to attempt to place the pieces on, we simply need to do some maths with the co-ordinates to find the squares in between the checker and the king.

The last thing we need to check (although the order doesn't matter) is whether the king can move out of check. To do this we can simply test all of the 8 possible king moves and as above we can put it through our algorithm, if it does not get us out of check (or moves us into another one) the algorithm will return false, the position will go back and we will try the next possibility. The board below shows the 8 possible moves for a king:



These can be represented in vector form as: (0, 1), (1, 1), (1, 0), (-1, 1), (0, -1), (-1, -1), (-1, 0), (1, -1).

This concludes the main logic to a game of standard chess.

2.6 Designing the interaction between boards

Although in the final game, I will certainly want the players to each use local machines. For the purpose of developing my project I will do most testing and development using local clients and servers so that I am able to test and debug it.

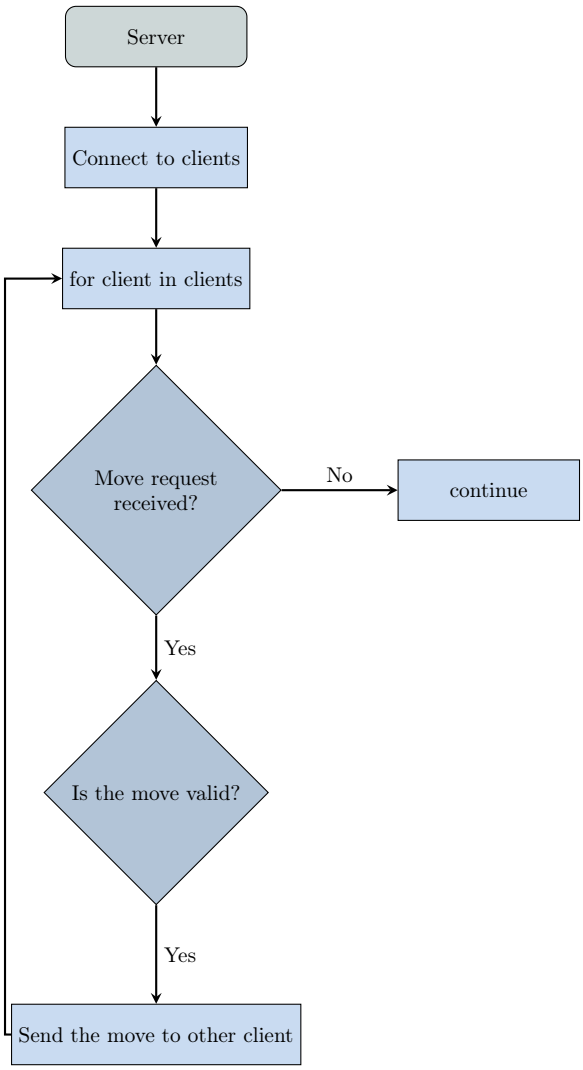
The first thing to design is a basic chess game between two players (white and black). The server, which both clients will connect to via socket, will be responsible for validating the moves made by each player and updating the other client's board if the move is valid. This way, both players will have access to the most current state of the game.

2.6.1 The Server

The server will need to do a number of things:

- Connect to the two clients
- Receive requests from the clients, process the request and return the validity of the move
- If a move is valid, send that move to the **other** client to update their board

Because of this the server will have most of the main chess logic. I plan to alternate between checking been move requests from each client, as it isn't possible to listen to both at the same time so it makes most sense to alternate.



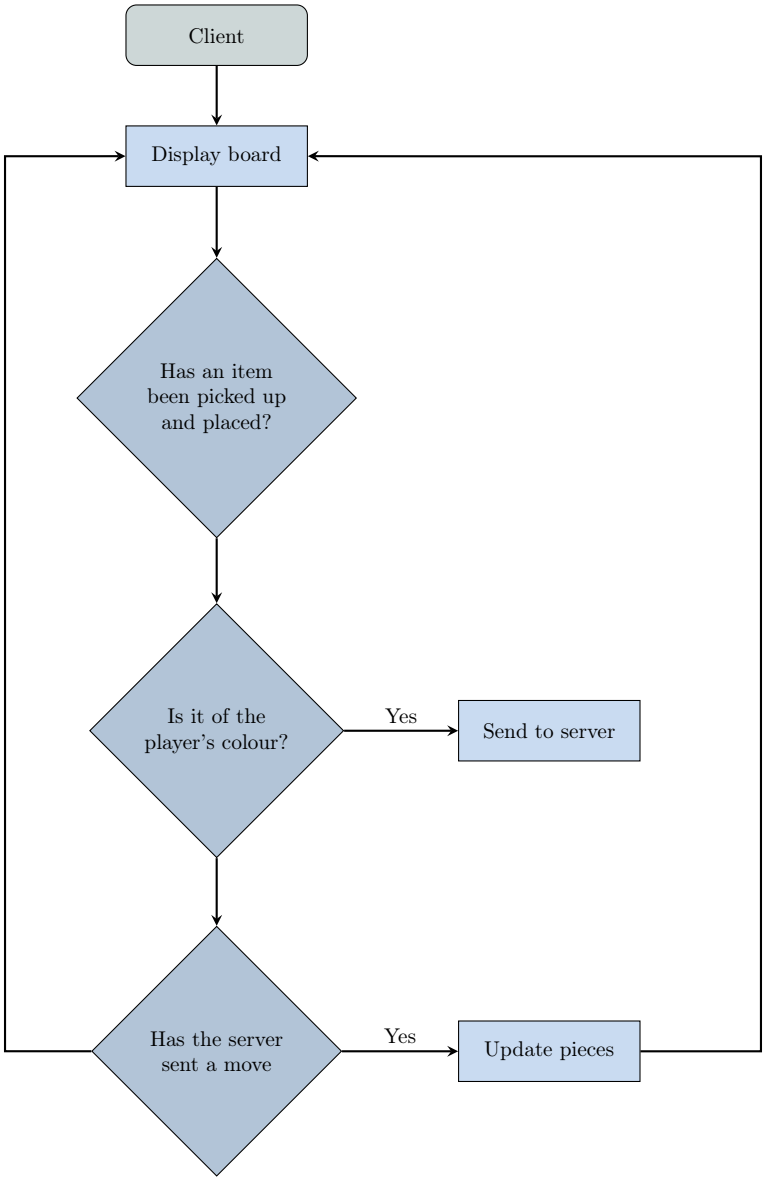
As you can see from the flowchart above, the server will constantly be awaiting requests. If the move being processed results in a checkmate, the server must send this information to both clients and end the game. This is not shown on the flowchart for the sake of brevity.

As mentioned above, the "Is move valid?" decision block, will make use of the logic designed in the **section above**.

2.6.2 The Clients

As the server is processing the movements, all the client needs to do is detect when a move is being attempted, and send the relevant data to the server. It

also needs to be able to receive information from the server when its opponent makes a move.



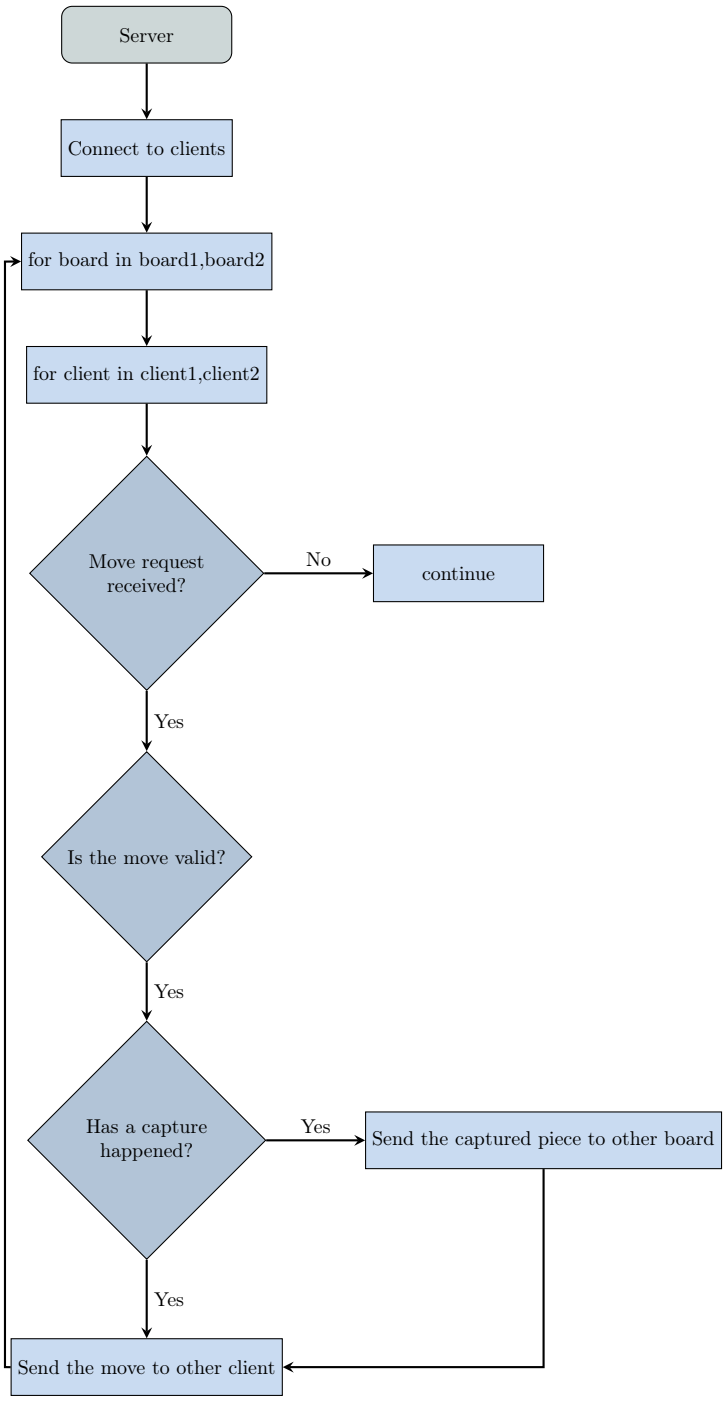
The clients will be largely the same, the only major difference is that one client will be able to send data if they are moving the black pieces and the other the white pieces.

The next step is to upscale this to allow for two boards that can play at the same time. This is rather simple and we essentially need to modify the flowchart of the **server**, so that it performs the actions for both boards.

As well as this I need to update the server to send pieces to the other board when a piece is captured. In conjunction with this, the client will need to be updated so that it can receive the new pieces and place them accordingly.

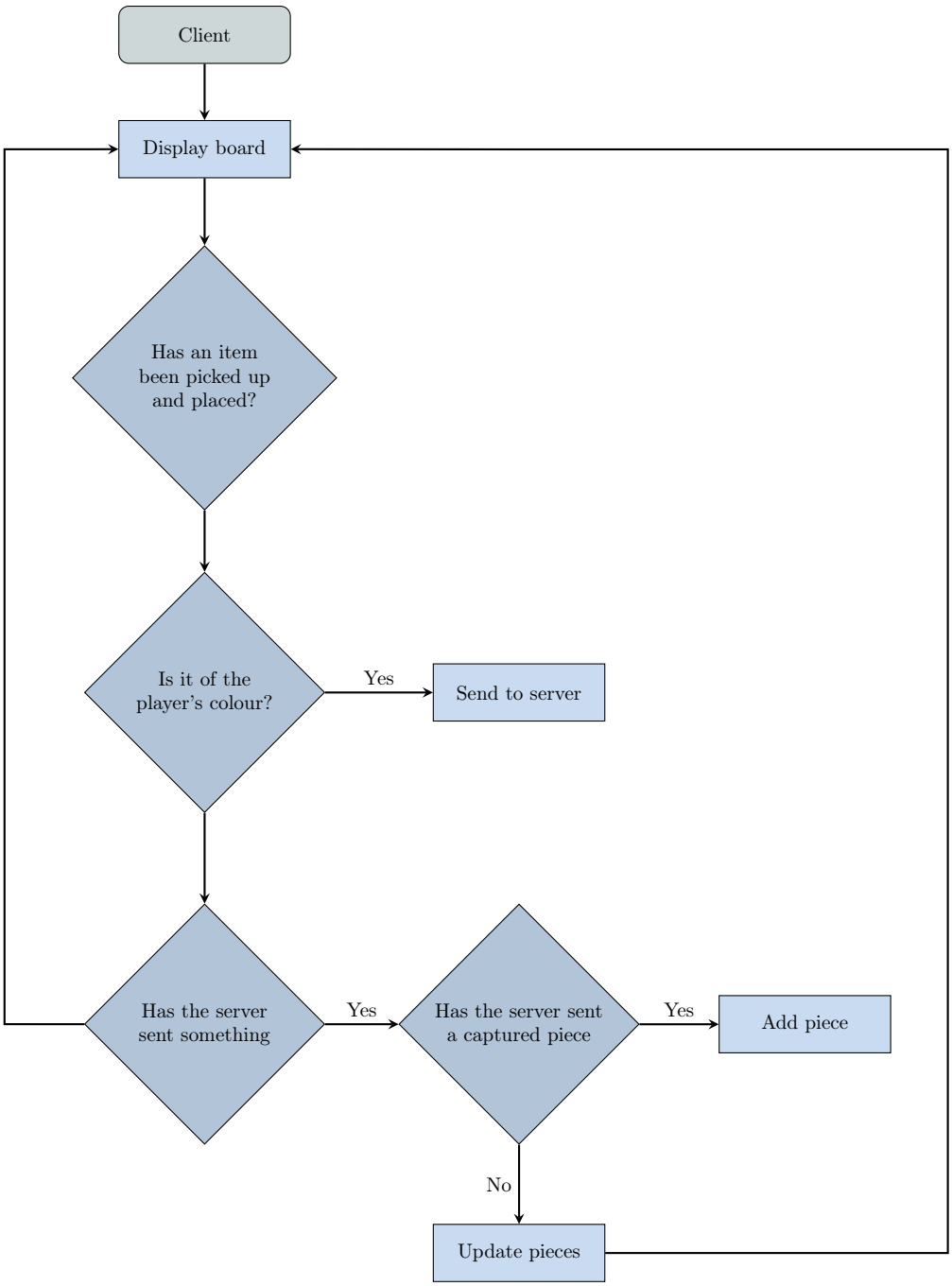
2.6.3 The Server Part 2

Below is the updated flowchart for the server. As you can see it now processes both boards as well as sending captured pieces to the other baord


Server
Connect to clients
for board in board1,board2
for client in client1,client2
Move request received?
No
continue
Yes
Is the move valid?
Yes
Has a capture happened?
Yes
Send the captured piece to other board
Yes
Send the move to other client

2.6.4 The Client Part 2

Below is the updated flowchart which includes the process for adding pieces that have been passed by the other board. The logic is simplified and in reality the process of adding a piece will require the creation of a new object.



As I won't be able to pass images over socket, I will simply send the name of the piece and add some methods to update the image and position. When a board

gets a new piece, I would like it to be on the left hand side of the board from the players perspective, and next to the correct rank depending on the piece they are. As it is possible to have more than 1 extra piece of the same type I will use a number to show how many pieces there are.

As I can't send images over socket I will use the following method to update the image of a piece using it's name.

As well as that I will add a method to update the position to the correct positions on the side of the board

Algorithm 4 update_image

```
function UPDATE_IMAGE(self)
    SET images TO { "wp": wpawn_image, "wq": wqueen_image, "wr": wrook_image, "wb": wbishop_image, "wn": wknight_image, "wk": wking_image, "bp": bpawn_image, "bq": bqueen_image, "br": brook_image, "bb": bbishop_image, "bn": bknight_image, "bk": bking_image }
    SET self.image TO images.get(self.name[:2],self.image)
end function
```

In reference to the number that displays how many pieces are available, although I could display it in **pygame** by drawing circles and numbers, I think it would be more straightforward and better looking for the user to simply "blit" images depending on how many are available.

To continue with my object oriented approach I plan to make a class for the "bubbles", that store the amount of pieces on a specific square (square here refers to the imaginary squares that new pieces will stay on).

I had not previously considered this when I presented the **classes** so here is an updated table:

Class Name	attribute name	Description
Piece	name	The name of the piece
Piece	xpos	The x-coordinate of the piece (square)
Piece	ypos	The y-coordinate of the piece (square)
Piece	Colour	Colour of piece
Piece	image	The image of the piece
Piece	placerx	The x-coordinate of the piece (pixel)
Piece	placery	The y-coordinate of the piece (pixel)
Piece	move_num	The amount of time the piece has been moved
Bubble	xpos	The y-coordinate of the bubble
Bubble	ypos	The y-coordinate of the bubble
Bubble	amount	Amount of pieces

This now concludes the main logic to the entire bughouse game.

2.7 Test Plan

Ideally for each test I want to include the following data:

- Normal Data
- Boundary Data
- Erroneous Data

For most of my tests I won't need to worry about the last two as most of my validation is handled by the fact that the user will only be able place pieces on the digital board so it is hard to input invalid data.

In order to test the individual **move_valid** functions for each piece I will use a range of inputs, some that will return True and other False. The main thing I can think of for possible erroneous data is trying to move a piece onto a "square" that is not on the board or trying to move a piece onto a square where a piece of the same colour already is. However, as this will apply to all pieces I think it makes more sense to take care of this in the main game loop when an attempted move is made.

To thoroughly test checkmate detection, I will create a range of test cases that include both checkmate and non-checkmate positions, as well as positions that are, in a sense, on the boundary of being in checkmate. I will present these positions along with the type of data they represent and, if necessary, an explanation for why they were included. By covering a diverse set of test cases, I can ensure that the checkmate detection function is accurate and reliable.

In terms of the pieces, I intend to create tests for the knight and queen moves. The reason for this is that the rook and bishop move-sets are simply subsets of the queen move-set, so if the queen move-set works then the bishop and rook move-sets should too. I am not testing the king's move-set as it is so simple there are not many possible scenarios and I will be able to easily check these manually. I have similar reasoning for not testing the pawn moves as there are so few possible moves, I can easily test them manually and gauge whether the function is working.

Because of the Object Oriented approach I have gone with, creating tests is not easy as I essentially have to create all the objects (pieces) to carry out a test. For that reason I am only creating tests for the necessary parts.

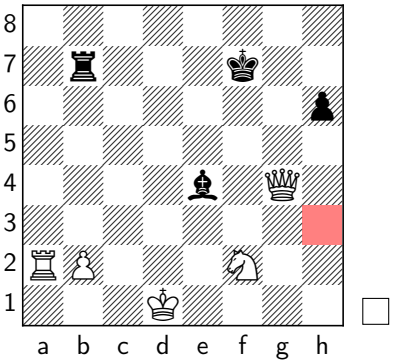
2.7.1 Testing Knight Moves

Below I will use a red highlighted square to show where the piece is trying to be moved.

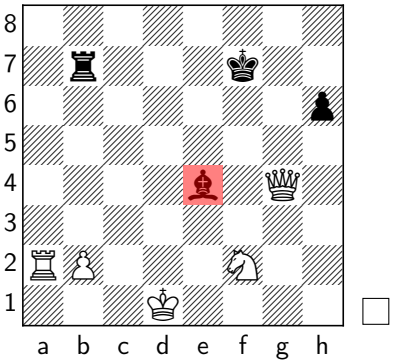
The test cases below cover all the possibilities that could come up:

- Moving to a free square that a knight can move to (valid)
- Moving to a square a knight can move to with an enemy piece (valid)

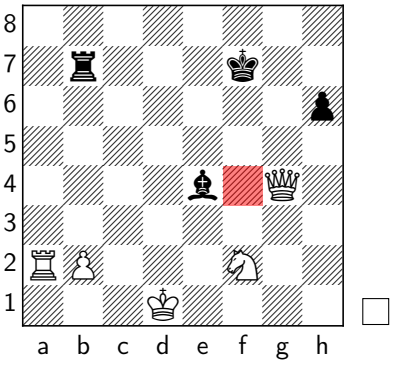
- Moving to a free square that a knight **can't** move to (invalid)
- Moving to a free square with a piece of the same colour (invalid)
- Moving to a square off the board (invalid)



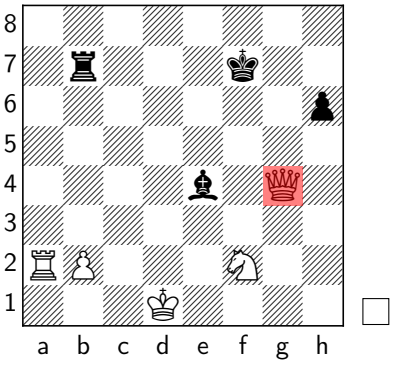
Test Case 1



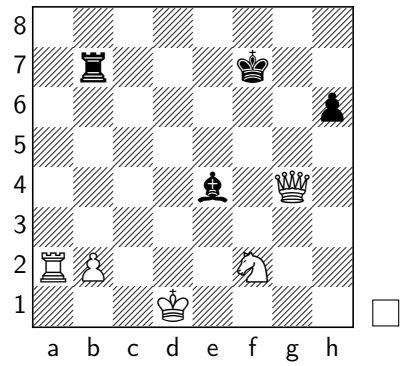
Test Case 2



Test Case 3



Test Case 4

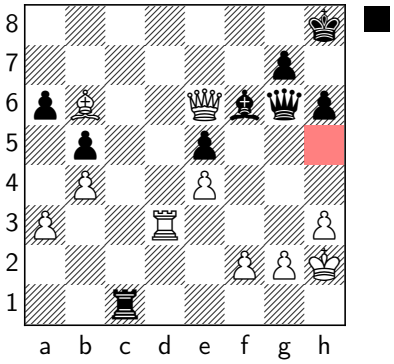


Test Case 5 - attempting to move to "e0"

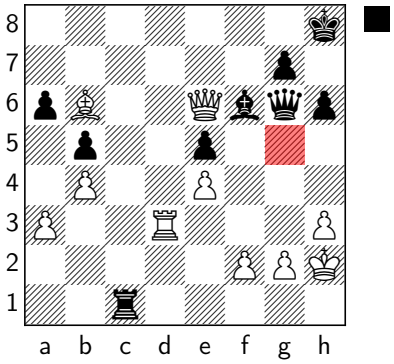
2.7.2 Testing Queen Moves

The test cases for the Queen covers very similar possibilities to the knight but incorporates the fact that the queen can move in two different ways (straight or diagonal):

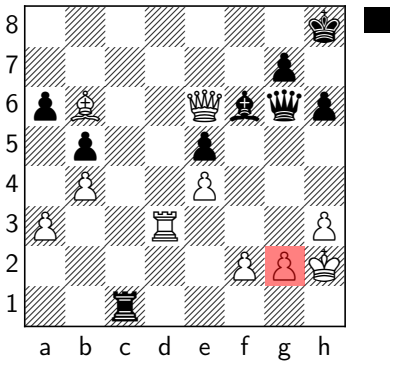
- Moving diagonally to a free square (valid)
- Moving straight to a free square (valid)
- Moving to a square with an enemy piece straight (valid)
- Moving to a square with an enemy piece diagonally (valid)
- Moving to a square with a team piece (invalid)
- Moving in the correct way but off the board (invalid)



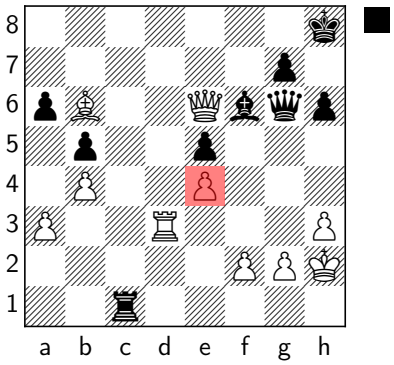
Test Case 1



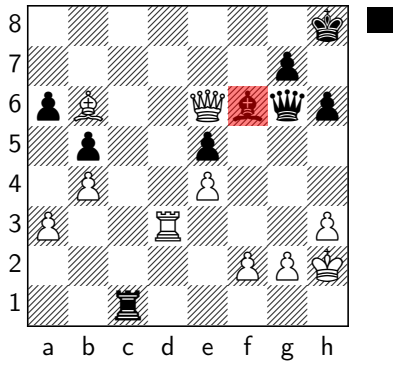
Test Case 2



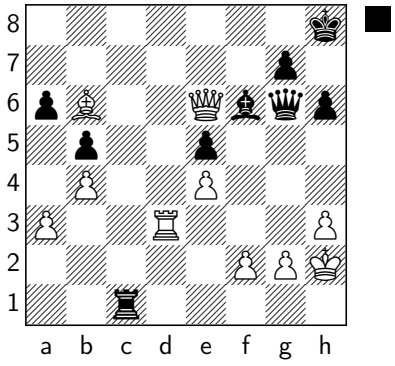
Test Case 3



Test Case 4



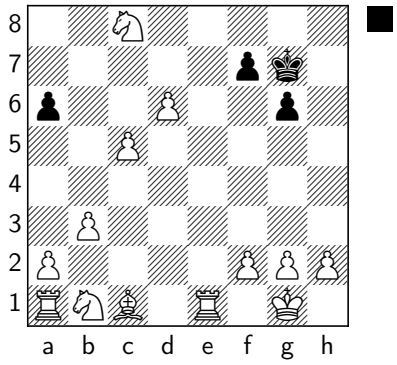
Test Case 5



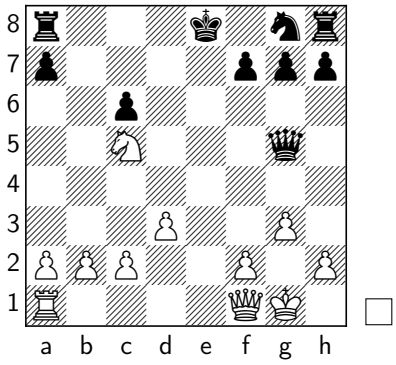
Test Case 6 - attempting to move to "i4"

2.7.3 Testing Checkmate

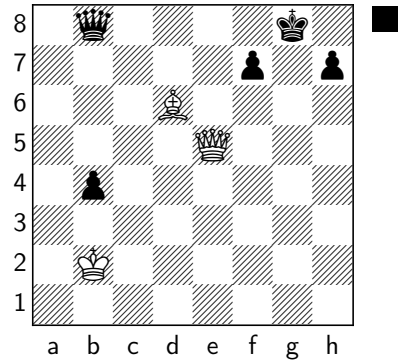
Normal test data (not checkmate or check):



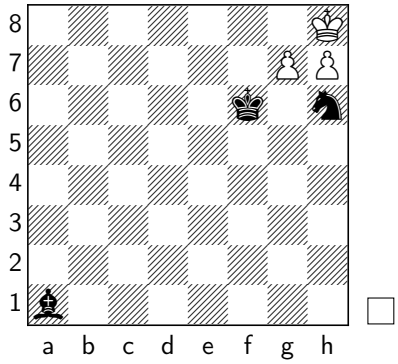
Test Case 1



Test Case 2

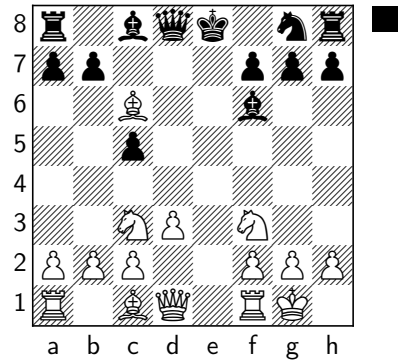


Test Case 3

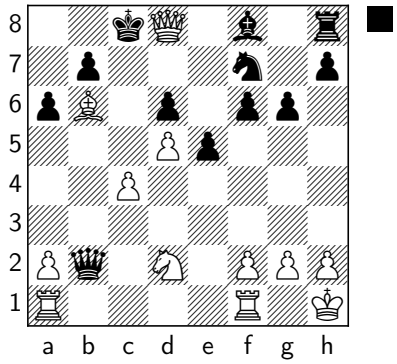


Test Case 4

Normal test data (not checkmate but in check):

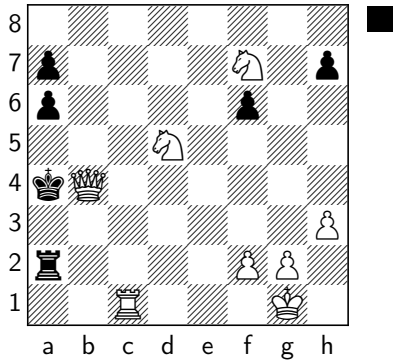


Test Case 5

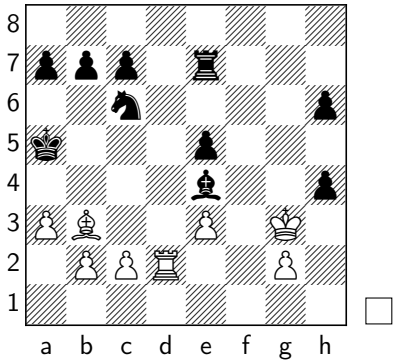


Test Case 6

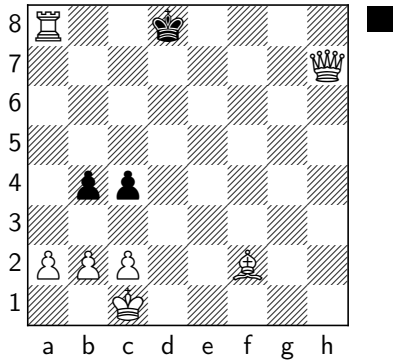
Normal test data (checkmate):



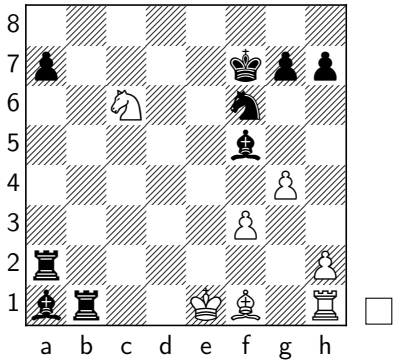
Test Case 7



Test Case 8

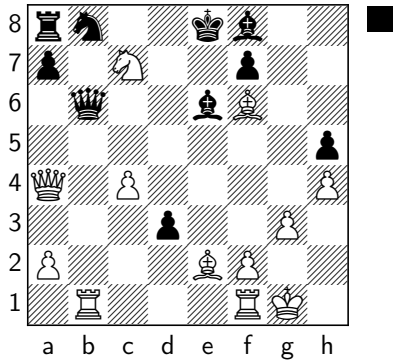


Test Case 9

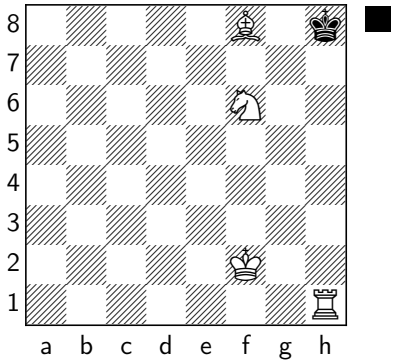


Test Case 10

Boundary test data (checkmate by discovered check):

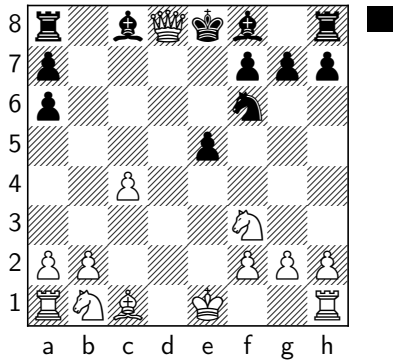


Test Case 11

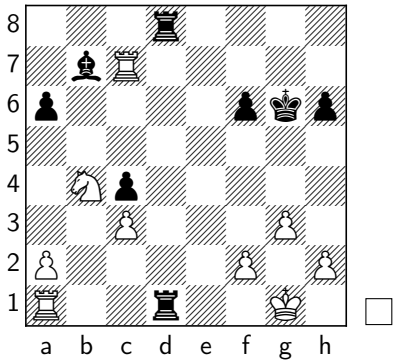


Test Case 12

Boundary test data (can only escape by capturing):

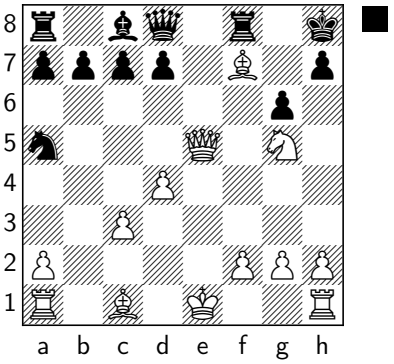


Test Case 13

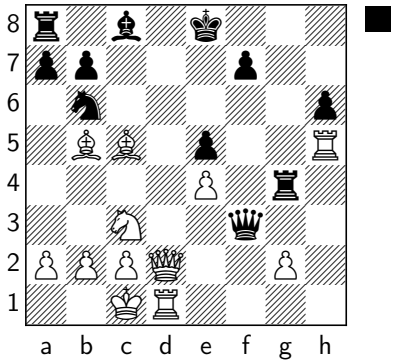


Test Case 14

Boundary test data (can only escape by blocking):

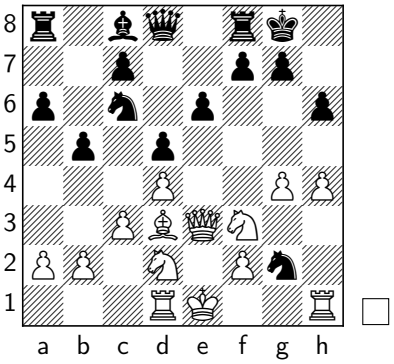


Test Case 15

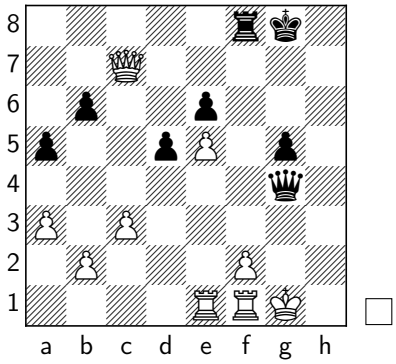


Test Case 16

Boundary test data (can only escape by moving king):



Test Case 17



Test Case 18

I included lots of test cases that cover many of the possible scenarios in order to thoroughly test the functionality and ensure that me code works correctly in all possible scenarios.

3 Developing the coded solution

3.1 Development Cycle 1

3.1.1 Aims

In this development cycle I aim to develop the base GUI and logic to a game of chess. This excludes the additional rules of Bughouse and will involve two players playing from one client.

3.1.2 Development

To begin with I coded the initialisation of the pygame screen as well as loading some images to pygame for the White King and Board. I also scaled the image down to the correct size for the board.

As well as this I created a list of lists that I can later use to view the positions of the pieces without relying on the GUI.

```
pygame.init()
screen = pygame.display.set_mode((1000, 800))
clock = pygame.time.Clock()
white = (255, 255, 255)
myfont = pygame.font.SysFont("Comic Sans MS", 30)

image_constant = (75, 75)
board_image = pygame.image.load(
    r"C:\Users\danie\PycharmProjects\ChessGit\Images\board.png")
wking_image = pygame.image.load(
    r"C:\Users\danie\PycharmProjects\ChessGit\Images\wking.png").
                                        convert_alpha()
wking_image = pygame.transform.scale(wking_image, image_constant)

board = [[" " for i in range(8)] for i in range(8)]
print("".join([f"\n{i}" for i in board]))
```

The next thing I did was create the class for the pieces:

```
class Piece:
    def __init__(self, name, xpos, ypos, colour, image=wking_image)
                                            :
        self.xpos = xpos
        self.ypos = ypos
        self.colour = colour
        self.name = name
        self.image = image
        self.placerx = 125 + self.xpos * 75
        self.placery = self.ypos * 75

    def info(self):
        print(self.xpos, self.ypos)
        pass

    def namer(self):
```

```
        print(self.name)

    def place(self):
        board[self.ypos - 1][self.xpos - 1] = self.name
```

As you can see, I have made the default of the image the White King Image, this means that for the time-being I can work on the movement of the pieces, without worrying about sourcing images for all the pieces.

The clear next step to me, is to create the objects themselves (the standard chess pieces).

```
wp = []
for i in range(1, 9):
    wp.append(Piece((f"wp{i}"), i, 7, "w"))
wp.append(Piece((f"wk"), 4, 8, "w"))
wp.append(Piece((f"wq"), 5, 8, "w"))
wp.append(Piece((f"wb1"), 3, 8, "w"))
wp.append(Piece((f"wb2"), 6, 8, "w"))
wp.append(Piece((f"wn1"), 2, 8, "w"))
wp.append(Piece((f"wn2"), 7, 8, "w"))
wp.append(Piece((f"wr1"), 1, 8, "w"))
wp.append(Piece((f"wr2"), 8, 8, "w"))
for i in wp:
    (i.info())
    i.place()

print("".join([f"\n{i}" for i in board]))

bp = []
for i in range(1, 9):
    bp.append(Piece((f"bp{i}"), i, 2, "b"))
bp.append(Piece((f"bk"), 5, 1, "w"))
bp.append(Piece((f"bq"), 4, 1, "w"))
bp.append(Piece((f"bb1"), 3, 1, "w"))
bp.append(Piece((f"bb2"), 6, 1, "w"))
bp.append(Piece((f"bn1"), 2, 1, "w"))
bp.append(Piece((f"bn2"), 7, 1, "w"))
bp.append(Piece((f"br1"), 1, 1, "w"))
bp.append(Piece((f"br2"), 8, 1, "w"))
```

Although the code for this is not very concise, there doesn't seem to be a much more concise way to create all the pieces, which have quite unique and "random" seeming positions.

The pawns on the other hand take up a row for each colour so I used a loop to create all of the pawns.

I am also going to create an overall list to contain both sets of pieces as I expect some of my algorithms will require me look through all the pieces not just on set, yet some may only require me to look through one set.

The next thing I need to do is create the main loop for the game. It seems sensible to run the game on a tick base so that it updates every tick and performs

necessary calculations, such as checking a move is valid.

```
run = True
while run:
    clock.tick(120)
    screen.fill(white)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

    #Here will be the main game loop logic

    pygame.display.update()
```

I have used a clock tick of 120, which means at most 120 frames will pass per second. Although most monitors do not have a refresh rate of 120, my home monitor does so I will be able to see the smoothest graphics as possible.

If I run into performance issues in the future I plan to reduce this to a lower value such as 60 to lower the amount of calculations.

In the main game loop, I need to display the board image each tick and also run through each piece to see if the player is trying to move it

```
    screen.blit(board_image, (200, 75))
    for i in ap:
        screen.blit(i.image, (i.placerx, i.placery))
        if event.type == pygame.MOUSEBUTTONDOWN and event.button ==
                                            1: #If the player has
                                           the left mouse button
                                           down
            x, y = pygame.mouse.get_pos() #Store the values of the
                                              mouse position
            for i in range(1, 9): # Converts the value of the mouse
                                               position into square
                                               co-ordinates
                if x > 200 + (i - 1) * 75 and x < 200 + i * 75:
                    newposx = i
                    break
            for i in range(1, 9):
                if y > 75 + (i - 1) * 75 and y < 75 + i * 75:
                    newposy = i
                    break
            for i in ap:
            #Runs through the pieces to see if the mouse is pressed
                                               whilst on a piece
                if i.xpos == newposx and i.ypos == newposy:
                    i.placerx = x
                    i.placery = y
                    item = i
        if pygame.mouse.get_pressed()[0] and item:
        #If the mouse button is pressed we update the placer co-
                                          ordinates of the piece so
                                           that the player can drag
                                           piecesr
            x, y = pygame.mouse.get_pos()
```

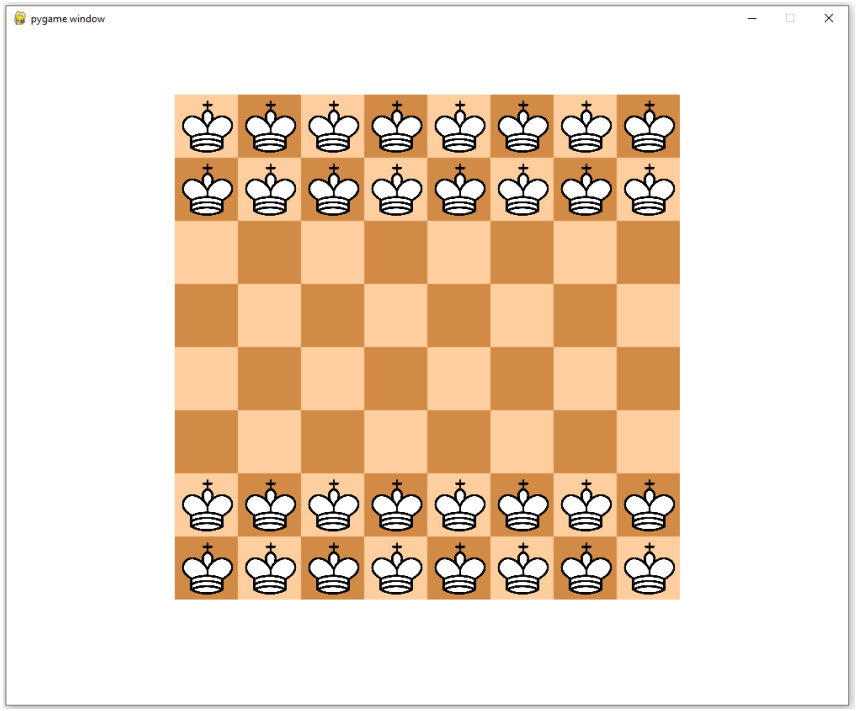
```
            item.placerx = x - 75 / 2
            item.placery = y - 75 / 2
        if not pygame.mouse.get_pressed()[0]:
            try: #If the left mouse button is not pressed down we
                                            try the following
                    x, y = pygame.mouse.get_pos()
                    for i in range(1, 9):
                        if x > 200 + (i - 1) * 75 and x < 200 + i *
                                                        75: #
                                                        Converts
                                                        the
                                                        position
                                                        into
                                                        square co
                                                        -ords so
                                                        that it "
                                                        snaps" to
                                                         the
                                                        square
                            xsquare = i
                            break
                    for i in range(1, 9):
                        if y > 75 + (i - 1) * 75 and y < 75 + i *
                                                        75:
                            ysquare = i
                            break
                    #Referring to item object so will only work if
                                                        item is not
                                                        None
                    #We will set item to None at the end of each
                                                        cycle of the
                                                        loop unless
                                                        the mouse
                                                        button is
                                                        pressed down
                    item.placerx = 125 + xsquare * 75
                    item.placery = ysquare * 75
                    for i in ap:
                        if i.xpos == xsquare and i.ypos == ysquare
                                                        and i !=
                                                        item:
                            ap.remove(i)
                    item.xpos = xsquare
                    item.ypos = ysquare
                    if xsquare == 0 or ysquare == 0:
                        item.placerx = 125 + newposx * 75
                        item.xpos = newposx
                        item.placery = newposy * 75
                        item.ypos = newposy
                except AttributeError:
                    pass
                except NameError:
                    pass
                item = None
```

The section above that converts co-ordinates to square coordinates when the

player lets go of a piece is quite length so it makes sense to me to replace it with a function and call the function from inside the game loop.

```
    def snapper(x, y):
    snapposx, snapposy = 0, 0
    for i in range(1, 9):
        if x > 200 + (i - 1) * 75 and x < 200 + i * 75:
            snapposx = i
            break
    for i in range(1, 9):
        if y > 75 + (i - 1) * 75 and y < 75 + i * 75:
            snapposy = i
            break
    return snapposx, snapposy
```

As you can see below, the code works. However, I haven't yet imported and associated images for each piece, which looks quite comedic. I will do that and then the next step is to add the movesets discussed in the design.



To source the images for the pieces I found a sprite sheet with a transparent background and separated them into separated pieces. I then loaded all the images into my file and changed the images I use for each piece.

Figure 1: Sprite Sheet

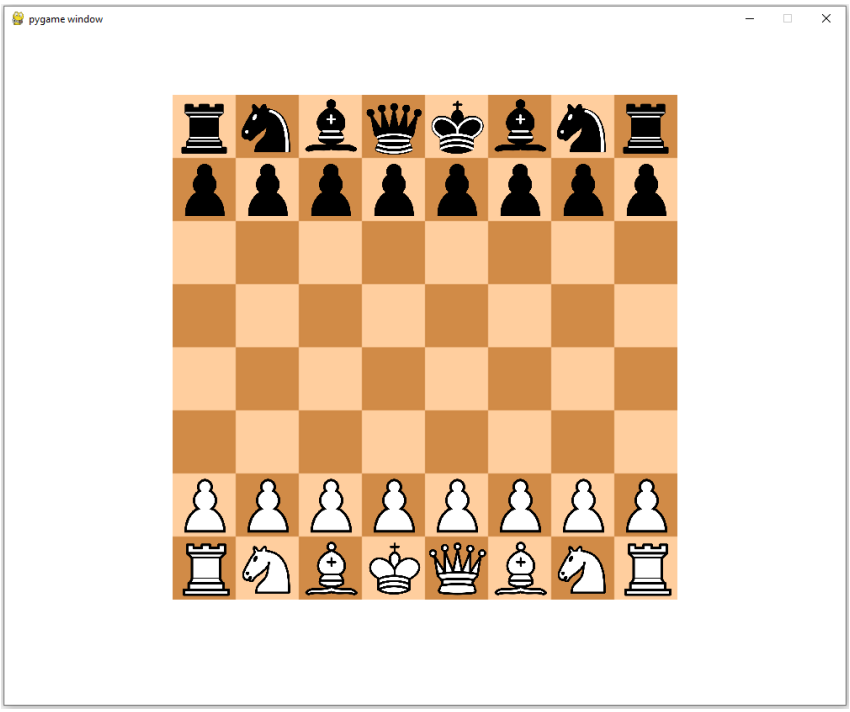
```
wp = []
for i in range(1, 9):
    wp.append(Piece((f"wp{i}"), i, 7, "w", wpawn_image))
wp.append(Piece((f"wk"), 4, 8, "w", wking_image))
wp.append(Piece((f"wq"), 5, 8, "w", wqueen_image))
wp.append(Piece((f"wb1"), 3, 8, "w", wbishop_image))
wp.append(Piece((f"wb2"), 6, 8, "w", wbishop_image))
wp.append(Piece((f"wn1"), 2, 8, "w", wknight_image))
wp.append(Piece((f"wn2"), 7, 8, "w", wknight_image))
wp.append(Piece((f"wr1"), 1, 8, "w", wrook_image))
wp.append(Piece((f"wr2"), 8, 8, "w", wrook_image))
for i in wp:
    (i.info())
    i.place()

print("".join([f"\n{i}" for i in board]))

bp = []
for i in range(1, 9):
    bp.append(Piece((f"bp{i}"), i, 2, "b", bpawn_image))
bp.append(Piece((f"bk"), 5, 1, "w", bking_image))
bp.append(Piece((f"bq"), 4, 1, "w", bqueen_image))
bp.append(Piece((f"bb1"), 3, 1, "w", bbishop_image))
bp.append(Piece((f"bb2"), 6, 1, "w", bbishop_image))
bp.append(Piece((f"bn1"), 2, 1, "w", bknight_image))
bp.append(Piece((f"bn2"), 7, 1, "w", bknight_image))
bp.append(Piece((f"br1"), 1, 1, "w", brook_image))
bp.append(Piece((f"br2"), 8, 1, "w", brook_image))
```

Unfortunately, loading and scaling the images is quite lengthy and for the moment I cannot see a more concise way of doing it. If I do find a cleaner way I will.

```
#Loads images
wqueen_image = pygame.image.load(r"Images\wqueen.png")
wbishop_image = pygame.image.load(r"Images\wbishop.png")
wknight_image = pygame.image.load(r"Images\wknight.png")
wrook_image = pygame.image.load(r"Images\wrook.png")
wpawn_image = pygame.image.load(r"Images\wpawn.png")

bking_image = pygame.image.load(r"Images\bking.png")
bqueen_image = pygame.image.load(r"Images\bqueen.png")
bbishop_image = pygame.image.load(r"Images\bbishop.png")
bknight_image = pygame.image.load(r"Images\bknight.png")
```

```
brook_image = pygame.image.load(r"Images\brook.png")
bpawn_image = pygame.image.load(r"Images\bpawn.png")

#Scales images
wking_image = pygame.transform.scale(wking_image, image_constant)
wqueen_image = pygame.transform.scale(wqueen_image, image_constant)
wbishop_image = pygame.transform.scale(wbishop_image,
                                       image_constant)
wknight_image = pygame.transform.scale(wknight_image,
                                       image_constant)
wrook_image = pygame.transform.scale(wrook_image, image_constant)
wpawn_image = pygame.transform.scale(wpawn_image, image_constant)

bking_image = pygame.transform.scale(bking_image, image_constant)
bqueen_image = pygame.transform.scale(bqueen_image, image_constant)
bbishop_image = pygame.transform.scale(bbishop_image,
                                       image_constant)
bknight_image = pygame.transform.scale(bknight_image,
                                       image_constant)
brook_image = pygame.transform.scale(brook_image, image_constant)
bpawn_image = pygame.transform.scale(bpawn_image, image_constant)
```

With this, the default set-up is looking a lot more normal.



Now I will implement the move-sets which I designed in the **Analysis Section**.

```
def piecethere(xsquare, ysquare):
    for i in ap: #returns if a piece is at a given square
        if i.xpos == xsquare and i.ypos == ysquare and i!=item:
            return True

    return False

def piecethereexclude(xsquare, ysquare):
    for i in ap: #returns if a piece of opposite colour is at a
                                        given square
        if i.xpos == xsquare and i.ypos == ysquare and i.name[0]!=
                                            item.name[0]:
            return True
    return False


def kingmoves(x, y, item): #returns if a king move is valid
    if abs(x) < 2 and abs(y) < 2 and not piecethereexclude(item.
                                        xpos+x,item.ypos+y):
        returner = True
    else:
        returner = False
    return returner

def queenmoves(x, y, startx, starty): #returns if a queen move is
                                    valid
    returner = True
    if abs(x) == abs(y) or x == 0 or y == 0:
        if abs(x) == abs(y): #if diagonal move
            for i in range(0, abs(x)+1):
                if x <0:
                    vectorx = startx - i
                elif x > 0:
                    vectorx = startx + i
                if y <0:
                    vectory = starty - i
                elif y > 0:
                    vectory = starty + i
                if abs(x) == i and piecethereexclude(vectorx,
                                                    vectory):
                    return True
                elif piecethere(vectorx, vectory):
                    returner = False
                    return returner
                else:
                    returner = True
        elif y == 0: #if horizontal move
            for i in range(1, abs(x)+1):
                if x > 0:
                    vectorx = startx + i
                if x < 0:
                    vectorx = startx - i
                if abs(x) == i and piecethereexclude(vectorx,starty
                                                    ):
                    return True
                elif piecethere(vectorx, starty):
                    returner = False
```

```
                    break
                else:
                    returner = True
        elif x == 0: #if vertical move
            for i in range(1, abs(y)+1):
                if y > 0:
                    vectory = starty + i
                if y < 1:
                    vectory = starty - i
                if abs(y) == i and piecethereexclude(startx,vectory
                                                     ):
                    return True
                elif piecethere(startx, vectory):
                    returner = False
                    break
                else:
                    returner = True
    else:
        returner = False
    return returner

def bishop_moves(x, y, startx, starty): #returns if a bishop move
                                    is valid
    if abs(x) == abs(y):
        for i in range(0, abs(x) + 1):
            if x < 0:
                vectorx = startx - i
            elif x > 0:
                vectorx = startx + i
            if y < 0:
                vectory = starty - i
            elif y > 0:
                vectory = starty + i

            if abs(x) == i and piecethereexclude(vectorx, vectory):
                return True
            elif piecethere(vectorx, vectory):
                returner = False
                return returner
            else:
                returner = True
    else:
        returner = False
    return returner

def knight_moves(x, y, startx, starty): #returns if a knight move
                                    is valid
    if (abs(x) == 1 and abs(y) == 2) or (abs(x) == 2 and abs(y) ==
                                         1):
        if piecethereexclude(startx + x, starty + y) or not
                                            piecethere(startx + x,
                                            starty + y):
            return True
    else:
        return False

def rook_moves(x, y, startx, starty): #returns if a rook move is
```

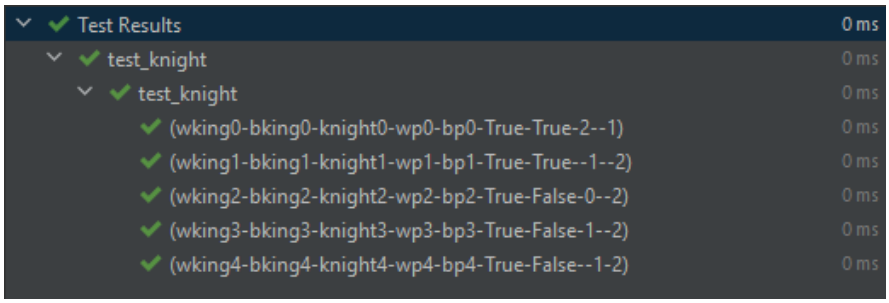
```
                                    valid
    if x == 0 or y == 0:
        if y == 0:
            for i in range(1, abs(x)+1):
                if x > 0:
                    vectorx = startx + i
                if x < 0:
                    vectorx = startx - i
                if abs(x) == i and piecethereexclude(vectorx,starty
                                                     ):
                    return True
                elif piecethere(vectorx, starty):
                    returner = False
                    break
                else:
                    returner = True
        elif x == 0:

            for i in range(1, abs(y)+1):
                if y > 0:
                    vectory = starty + i
                if y < 1:
                    vectory = starty - i
                if abs(y) == i and piecethereexclude(startx,vectory
                                                     ):
                    return True
                elif piecethere(startx, vectory):
                    returner = False
                    break
                else:
                    returner = True
    else:
        returner = False
    return returner

def white_pawn_moves(x, y, startx, starty, first): #returns if a
                                    white pawn move is valid
    if x == 0 and y == -1 and not piecethere(startx, starty - 1):
        return True
    elif abs(x)==1 and y == -1 and piecethereexclude(startx + x,
                                       starty - 1):
        return True
    elif x == 0 and y == -2 and not piecethere(startx + x, starty -
                                       1) and first == 0:
        return True
    else:
        return False

def black_pawn_moves(x, y, startx, starty, first): #returns if a
                                    black pawn move is valid
    if x == 0 and y == 1 and not piecethere(startx, starty + 1):
        return True
    elif abs(x)==1 and y == 1 and piecethereexclude(startx + x,
                                       starty + 1):
        return True
    elif x == 0 and y == 2 and not piecethere(startx + x, starty +
                                       1) and first == 0:
```

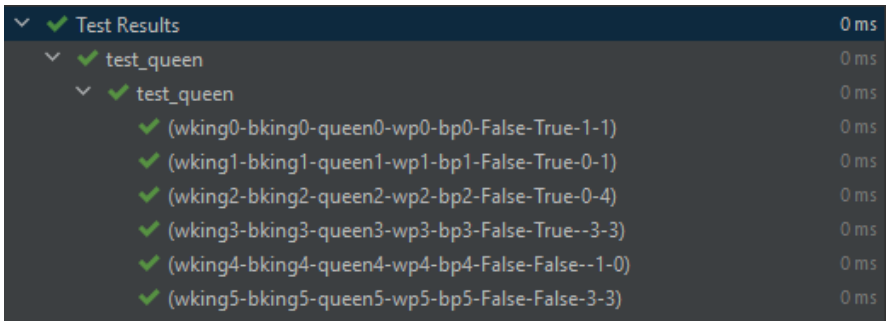
```
        return True
    else:
        return False
```

As well as implementing the functions designed earlier, I have also coded two functions **piecethere()** and **piecethereexclude()**. These two functions are extremely useful to see whether a move is valid because I piece can never move to a square with a piece of it's own colour and pawns can only move diagonally if a piece of opposite colour is there.

For that reason, I have made these both functions to avoid unnecessary repetition in my code.

At this stage it makes sense to test these functions with the **test cases** I planned in the design. The good news is that all tests were passed straight away, whcih is great news.





Now I will implement this at the end of the main game loop so that a move is only allowed if the relevant function returns Valid, otherwise the piece will be sent back to the square it was picked up from.

Now I am going to implement the turns, so that a player can only make a move on their turn. This should be quite a straightforward addition.

```
    if item.name[0] == "w" and move == True:
```

```
    displacex = abs(newposx - xsquare)  #displacex and displacey represent absolute vector of move
    displacey = abs(newposy - ysquare)
    movevalid = True
    tempx, tempy = item.xpos, item.ypos
    item.xpos, item.ypos = xsquare, ysquare
    if item.name[1] == "k":
        movevalid = kingmoves(xsquare-item.xpos, ysquare-item.ypos,item)
    elif item.name[1] == "q":
        movevalid = queenmoves(-(newposx - xsquare),ysquare - newposy, newposx, newposy)
    elif item.name[1] == "b":
        movevalid = bishop_moves(-(newposx - xsquare),ysquare - newposy, newposx, newposy)
    elif item.name[1] == "n":
        movevalid = knight_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy)
    elif item.name[1] == "r":
        movevalid = rook_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy)
    elif item.name[0:2] == "wp":
        movevalid = white_pawn_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item.move_num
                                                                                    )
        if movevalid:
            item.move_num += 1
    elif item.name[0:2] == "bp":
        movevalid = black_pawn_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item.move_num
                                                                                    )
        if movevalid:
            item.move_num += 1
    else:
        movevalid = True
    if not movevalid: #if the move is valid it returns the piece to its old position
        item.placerx = 125 + tempx * 75
        item.xpos = tempx
        item.placery = tempy * 75
        item.ypos = tempy
    if movevalid:
        for i in ap: #removes a captured piece
            if i.xpos == xsquare and i.ypos == ysquare and i != item and i.name[0]!=item.name[0]:
                ap.remove(i)
        item.xpos = xsquare
        item.ypos = ysquare
```

```
            turn = True
        elif item.name[0] == "b" and move == False:
            turn = True
```

I have added in the lines above in the game loop and updated the later lines to be:

```
    if not movevalid or not turn:
```

and

```
    if movevalid and turn:
```

Now I will add a function to detect check, this will be used to constrain pieces against moving if it causes them to stay in check or become in check.

```
    def check_checker(wp, bp, wking, bking):
        if move:
            pieces = bp
            king = wking
        else:
            pieces = wp
            king = bking
        for item in pieces:
            if item.name[1] == "k":
                movevalid = False
            elif item.name[1] == "q":
                movevalid = queenmoves(king.xpos - item.xpos, king.ypos - item.ypos, item.xpos, item.ypos)
                if movevalid:
                    return movevalid
            elif item.name[1] == "b":
                movevalid = bishop_moves(king.xpos - item.xpos, king.ypos - item.ypos, king.xpos, king.ypos)
                if movevalid:
                    return movevalid
            elif item.name[1] == "n":
                movevalid = knight_moves(item.xpos - king.xpos, item.ypos - king.ypos, king.xpos, king.ypos)
                if movevalid:
                    return movevalid
            elif item.name[1] == "r":
                movevalid = rook_moves(item.xpos - king.xpos, item.ypos - king.ypos, king.xpos, king.ypos)
                if movevalid:
                    return movevalid
            elif item.name[0:2] == "wp":
                movevalid = white_pawn_moves(item.xpos - king.xpos, item.ypos - king.ypos, king.xpos, king.
                                                                    ypos, item.move_num)
                if movevalid:
                    return movevalid
            elif item.name[0:2] == "bp":
                movevalid = black_pawn_moves(item.xpos - king.xpos, item.ypos - king.ypos, king.xpos, king.
                                                                    ypos, item.move_num)
                if movevalid:
                    return movevalid
            else:
                return False
```

At this point, I noticed a bug that was going to cause the check detection to give incorrect outputs. The reason for this is that I am simulating moves in the **check_checker()** function. This means that I cannot use the **item** variable in the functions that return the validity of legal moves because I am no longer only using the functions for the piece that has been picked up. This is because the **piecethere()** and **piecethereexclude()** functions need to account for themselves being on that square.

To fix this, I added a new parameter to most of my functions. When running the functions, I now pass in the relevant pieces. Although most functions don't use the **item**, and rather the raw moves, it is needed to pass to the functions mentioned above. This allows the functions to properly account for the simulated moves when determining the validity of legal moves.

Below are the updated function definitions:

```
    def piecethere(xsquare, ysquare, compare):
    def piecethereexclude(xsquare, ysquare, compare):
    def queenmoves(x, y, startx, starty, queen):
    def bishop_moves(x, y, startx, starty, bishop):
    def knight_moves(x, y, startx, starty, knight):
```

```
    def rook_moves(x, y, startx, starty, rook):
    def white_pawn_moves(x, y, startx, starty, first, wpa):
    def black_pawn_moves(x, y, startx, starty, first, bpa):
```

In the main game loop, I pass **item** into the relevant function. In **check_checker()** I pass in **piece** (the piece that I am simulating a move on).

It looks like the check detection is working as intended because the game is preventing players from moving out of check. However, it's not allowing players to make any moves at all when they're in check, which is certainly not the intended behavior.

To address this issue, I restructured some of the logic in the main game loop. In this new iteration, I update a piece's position regardless of whether the move is illegal. If the move is illegal I revert the position back to the previous state.

Previously, the piece's position was only updated if the move was valid, but this new approach allows for easier detection of check during simulation, particularly when trying to escape from check.

```
    tempx = item.xpos
    tempy = item.ypos
    item.xpos = xsquare #Piece's position updated striaght away
                                        when move attempted
    item.ypos = ysquare
    for i in ap:
        if i.xpos == xsquare and i.ypos == ysquare and i.name[0] !=
                                            item.name[0]:
            ap.remove(i)
            if i.name[0] == "w":
                wp.remove(i)
            else:
                bp.remove(i)
            tempitem = i
```

```
    if not movevalid or not turn:
        item.placerx = 125 + tempx * 75 #Position reverted to
                                          original if illegal move
        item.xpos = tempx
        item.placery = tempy * 75
        item.ypos = tempy
        if i:
            ap.append(tempitem)
            if tempitem.name[0] == "w":
                wp.append(tempitem)
            else:
                bp.append(tempitem)
    if movevalid and turn:
        move = not move
```

As soon as I played around with the board, having fixed the previous bug, I noticed that pawns could not capture anymore.

It quickly became clear to me that this bug was a direct result of the change I had just made. Because the function that checks if a pawn move is legal is run after the move is carried out, it does not allow for capturing as there is no piece on the square that the player is attempting to move to.

To fix this issue, I updated the main game loop to save the "captured" piece as an object. I also created a new function that checks if there is a piece on the square that the player is trying to move to, allowing for capturing again.

Along with these changes, I had to update some other small bits of syntax. I won't show these changes as they are all minor.

I made one other significant change to the **check_checker()** function. I replaced the variable **movevalid** with **checktake**, because I wanted a new variable to store the validity of simulated moves, rather than updating the **movevalid** variable for the actual move being attempted.

```
    def takenpiecechecker(xsquare, ysquare, compare):
    if takenpiece.xpos == xsquare and takenpiece.ypos == ysquare
                                        and takenpiece.name ==
                                        compare:
        return True
    else:
        takenpiece = None
        return False
```

The updated function to check the legality of a pawn move (same changes for the black pawn):

```
    def white_pawn_moves(x, y, startx, starty, first, wpa,
                                        takenpiece):
        if x == 0 and y == -1 and not piecethere(startx, starty - 1
                                            , wpa):
            return True
        elif abs(x)==1 and y == -1 and takenpiecechecker(takenpiece
                                            , startx + x, starty - 1,
                                             wpa):
            return True
        elif x == 0 and y == -2 and not piecethere(startx + x,
                                            starty - 1, wpa) and
                                            first == 0:
            return True
```

The next step now is to begin the implementation of the **checkmate_checker()**. When I began to do this I quickly realised I was going to need to use the piece movesets again, when generating pseudo-legal moves. Because of this I decided to make a function that refers pieces to their movesets. Previously I was just doing this in the main game loop, but instead of repeating this it would be better to make it into a function.

```
def move_valid(item,xsquare,ysquare,wp,bp,wking,bking,newposx,newposy):
    if item.name[1] == "k":
        movevalid = king_moves(xsquare - item.xpos, ysquare - item.ypos, item) and not check_checker(wp, bp,
                                                                                                     wking,bking)
    elif item.name[1] == "q":
        movevalid = queen_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item) and not
                                                                                                     check_checker(wp, bp, wking, bking)
    elif item.name[1] == "b":
        movevalid = bishop_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item) and not
                                                                                                     check_checker(wp, bp, wking, bking)
    elif item.name[1] == "n":
        movevalid = knight_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item) and not
                                                                                                     check_checker(wp, bp, wking, bking)
    elif item.name[1] == "r":
        movevalid = rook_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item) and not
                                                                                                     check_checker(wp, bp, wking, bking)
    elif item.name[0:2] == "wp":
        movevalid = white_pawn_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item.move_num
                                                                                                     , item,takenpiece) and not check_checker(wp,
                                                                                                     bp, wking, bking)
        if movevalid:
            item.move_num += 1
            tempitem = None
    elif item.name[0:2] == "bp":
        movevalid = black_pawn_moves(-(newposx - xsquare), ysquare - newposy, newposx, newposy, item.move_num
                                                                                                     , item,takenpiece) and not check_checker(wp,
                                                                                                     bp, wking, bking)
        if movevalid:
            item.move_num += 1
            tempitem = None
    else:
        movevalid = True
```

As you can see, I also changed the names of the piece moveset functions to have underscores between them as that is the correct PEP, and more consistent with the rest of my code. In a future iteration I hope to update my code so the pieces have an attribute which is their moveset.

In accordance with this change, I updated the main game loop to refer to the new function.

After making this change, I can now begin to make the **checkmate_checker** function. My first "draft" looked like this:

```
    def checkmate_checker(wp, bp, wking, bking, checking_pieces):
        if move:
            pieces = bp
            king = wking
        else:
            pieces = wp
            king = bking

        #Check if king can move out of check

        for checker in checking_pieces:
            if checker.name[1] = "b":
                #check bishop ray
            elif checker.name[1] = "r":
                #check rook ray
            elif checker.name[1] = "q":
                #check queen rays
```

```
            elif checker.name[1] = "n":
                #check if knight can be captured
            elif checker.name[1] = "p":
                #check if piece can be captured
```

Now I will begin to write the code for the individual pieces that could be checking, starting with the bishop.

```
if checker.name[1] = "b":
        for i in range(0,wking.xpos-checker.xpos):
            for piece in pieces:
                if move_valid(piece, checker.xpos+i, checker.ypos+i,wp,bp,wking,bking,piece.xpos,piece.
                                                                                    ypos) and not check_checker(wp, bp
                                                                                    , wking, bking):
                    checkmate = False
        #For all the squares between the checking bishop and the king in check, if a piece can block or
                                                                                take the bishop then it is not checkmate
```

The initial implementation demonstrated the potential of the concept, but it was limited to the scenario where the white king was in check. To make the code more applicable to both sides, I modified the first for loop to use the variable 'king.xpos' instead of 'wking.xpos'. Additionally, I used the absolute difference in the x positions to ensure that the for loop would execute correctly, even if the white king was positioned below the checking piece.

```
 elif checker.name[1] == "r":
        if checker.xpos - king.xpos == 0:
            for i in range(0,abs(king.ypos-checker.ypos)):
                for piece in pieces:
                    if move_valid(piece,checker.xpos, checker.ypos+i,wp,bp,wking,bking,piece.xpos,piece.
                                                                                    ypos) and not check_checker(wp
                                                                                    ,bp,wking,bking):
                        checkmate = False
        if checker.ypos - king.ypos == 0:
            for i in range(0,abs(king.xpos-checker.xpos)):
                for piece in pieces:
                    if move_valid(piece,checker.xpos+i, checker.ypos,wp,bp,wking,bking,piece.xpos,piece.
                                                                                    ypos) and not check_checker(wp
                                                                                    ,bp,wking,bking):
                        checkmate = False
```

```
elif checker.name[1] == "q":
        if checker.xpos - king.xpos == 0:
            for i in range(0,abs(king.ypos-checker.ypos)+1):
                for piece in pieces:
                    if move_valid(piece,checker.xpos, checker.ypos+i,wp,bp,wking,bking,piece.xpos,piece.
                                                                                    ypos) and not check_checker(wp
                                                                                    ,bp,wking,bking):
                        checkmate = False
        if checker.ypos - king.ypos == 0:
            for i in range(0,wking.xpos-checker.xpos):
                for piece in pieces:
                    if move_valid(piece,checker.xpos+i, checker.ypos,wp,bp,wking,bking,piece.xpos,piece.
                                                                                    ypos) and not check_checker(wp
                                                                                    ,bp,wking,bking):
                        checkmate = False
```

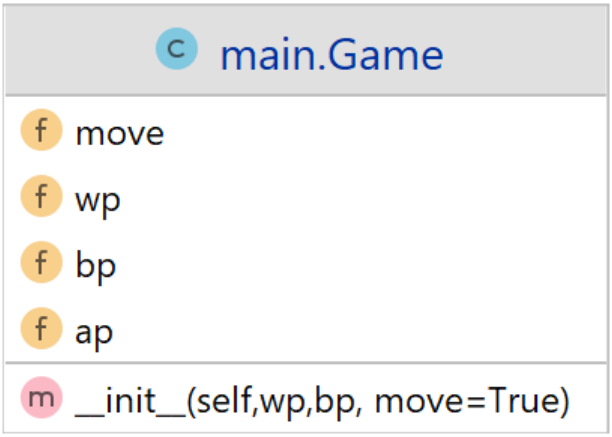
At this stage, I haven't yet implemented detection for a queen checking on a diagonal but I will carry out some testing first as I have written quite a lot of code without testing, so it makes sense to do this now.

During initial manual testing, I was able to easily identify and fix any bugs that were present. However, as the bugs have become less apparent, I will make sure to use my unit tests with the test data to continue finding and resolving any remaining issues.

When I tried to implement the tests that I designed for checkmate in my design section I realised it was difficult as I essentially need to pass in a whole game to test a position. In order to make testing work better I've decided to make the game an object and make the functions more object oriented. This involves passing in the game to most functions.

```
  class Game:
    def __init__(self,wp,bp, move):
        self.wp = wp
        self.bp = bp
        self.ap = wp + bp
        self.move = move
```

Here is the class diagram:



Game Class diagram

```
@pytest.mark.parametrize("wking,bking,wp, bp, expected_result, move
                                        ", [
    (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 1
        Piece(f"bk", 7, 2, "w", bking_image),
        [Piece(f"wk", 7, 8, "w", wking_image),
         Piece(f"wb1", 3, 8, "w", wbishop_image),
         Piece(f"wn1", 2, 8, "w", wknight_image),
```

```
        Piece(f"wn2", 3, 1, "w", wknight_image),
        Piece(f"wr1", 1, 8, "w", wrook_image),
        Piece(f"wr2", 5, 8, "w", wrook_image),
        Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 2, 6, "w", wpawn_image),
        Piece(f"wp3", 3, 4, "w", wpawn_image),
        Piece(f"wp4", 4, 3, "w", wpawn_image),
        Piece(f"wp5", 6, 7, "w", wpawn_image),
        Piece(f"wp6", 7, 7, "w", wpawn_image),
        Piece(f"wp7", 8, 7, "w", wpawn_image)],
       [Piece(f"bk", 7, 2, "b", bking_image),
        Piece(f"bp1", 6, 2, "b", bpawn_image),
        Piece(f"bp2", 7, 3, "b", bpawn_image),
        Piece(f"bp3", 1, 3, "b", bpawn_image)],
       False,
       False),
   (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 2
       Piece(f"bk", 5, 1, "b", bking_image),
       [Piece(f"wk", 7, 8, "w", wking_image),
        Piece(f"wp1", 1, 7, "w", wpawn_image),
       Piece(f"wp2", 2, 7, "w", wpawn_image),
       Piece(f"wp3", 3, 7, "w", wpawn_image),
       Piece(f"wp4", 4, 6, "w", wpawn_image),
       Piece(f"wp5", 6, 7, "w", wpawn_image),
       Piece(f"wp6", 7, 6, "w", wpawn_image),
       Piece(f"wp7", 8, 7, "w", wpawn_image),
       Piece(f"wq", 6, 8, "w", wqueen_image),
       Piece(f"wr1", 1, 8, "w", wrook_image),
       Piece(f"wn1", 3, 4, "w", wpawn_image),
    ],
    [  Piece(f"bk", 5, 1, "b", bking_image),
       Piece(f"bp1", 1, 2, "b", bpawn_image),
       Piece(f"bp2", 3, 3, "b", bpawn_image),
       Piece(f"bp3", 6, 2, "b", bpawn_image),
       Piece(f"bp4", 7, 2, "b", bpawn_image),
       Piece(f"bp5", 8, 2, "b", bpawn_image),
       Piece(f"bq", 7, 4, "b", bqueen_image),
       Piece(f"br1", 1, 1, "b", brook_image),
       Piece(f"br2", 8, 1, "b", brook_image),
       Piece(f"bn1", 7, 1, "b", bpawn_image)],
    False,
    True
    ),
   (Piece(f"wk", 2, 7, "w", wking_image), #TEST CASE 3
       Piece(f"bk", 7, 1, "b", bking_image),
       [Piece(f"wk", 2, 7, "w", wking_image),
       Piece(f"wq", 5, 4, "w", wqueen_image),
       Piece(f"wb1", 4, 3, "w", wbishop_image),
    ],
    [  Piece(f"bk", 7, 1, "b", bking_image),
       Piece(f"bp1", 2, 5, "b", bpawn_image),
       Piece(f"bp2", 6, 2, "b", bpawn_image),
       Piece(f"bp3", 8, 2, "b", bpawn_image),
       Piece(f"bq", 2, 1, "b", bqueen_image)],
    False,
    False
   ),
```

```
    (Piece(f"wk", 8, 1, "w", wking_image), #TEST CASE 4
        Piece(f"bk", 6, 3, "b", bking_image),
        [Piece(f"wk", 8, 1, "w", wking_image),
         Piece(f"wp1", 8, 2, "w", wpawn_image),
        Piece(f"wp2", 7, 2, "w", wpawn_image)],
        [Piece(f"bk", 6, 3, "b", bking_image),
        Piece(f"bb1", 1, 8, "b", bbishop_image),
        Piece(f"bn1", 8, 3, "b", bknight_image)],
        False,
        True),
    (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 5
        Piece(f"bk", 5, 1, "b", bking_image),
        [Piece(f"wk", 7, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 2, 7, "w", wpawn_image),
        Piece(f"wp3", 3, 7, "w", wpawn_image),
        Piece(f"wp4", 4, 6, "w", wpawn_image),
        Piece(f"wp5", 6, 7, "w", wpawn_image),
        Piece(f"wp6", 7, 7, "w", wpawn_image),
        Piece(f"wp7", 8, 7, "w", wpawn_image),
        Piece(f"wq", 4, 8, "w", wqueen_image),
        Piece(f"wr1", 1, 8, "w", wrook_image),
        Piece(f"wr2", 6, 8, "w", wrook_image),
        Piece(f"wn1", 3, 6, "w", wpawn_image),
        Piece(f"wn2", 6, 6, "w", wpawn_image),
         Piece(f"wb1", 3, 8, "w", wbishop_image),
         Piece(f"wb2", 3, 3, "w", wbishop_image)
     ],
     [  Piece(f"bk", 5, 1, "b", bking_image),
        Piece(f"bp1", 1, 2, "b", bpawn_image),
        Piece(f"bp2", 2, 2, "b", bpawn_image),
        Piece(f"bp3", 3, 4, "b", bpawn_image),
        Piece(f"bp4", 6, 2, "b", bpawn_image),
        Piece(f"bp5", 7, 2, "b", bpawn_image),
        Piece(f"bp6", 8, 2, "b", bpawn_image),
        Piece(f"bq", 4, 1, "b", bqueen_image),
        Piece(f"br1", 1, 1, "b", brook_image),
        Piece(f"br2", 8, 1, "b", brook_image),
        Piece(f"bn1", 7, 1, "b", bpawn_image),
        Piece(f"bb1", 3, 1, "b", bbishop_image),
        Piece(f"bb2", 6, 3, "b", bbishop_image)],
     False,
     False
    ),
    (Piece(f"wk", 8, 8, "w", wking_image), #TEST CASE 6
        Piece(f"bk", 3, 1, "b", bking_image),
        [Piece(f"wk", 8, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 3, 5, "w", wpawn_image),
        Piece(f"wp3", 4, 4, "w", wpawn_image),
        Piece(f"wp4", 6, 7, "w", wpawn_image),
        Piece(f"wp5", 7, 7, "w", wpawn_image),
        Piece(f"wp6", 8, 7, "w", wpawn_image),
        Piece(f"wq", 4, 1, "w", wqueen_image),
        Piece(f"wr1", 1, 8, "w", wrook_image),
        Piece(f"wr2", 6, 8, "w", wrook_image),
        Piece(f"wn1", 4, 7, "w", wpawn_image),
```

```
        Piece(f"wb1", 2, 3, "w", wbishop_image),
    ],
    [  Piece(f"bk", 3, 1, "b", bking_image),
       Piece(f"bp1", 1, 3, "b", bpawn_image),
       Piece(f"bp2", 2, 2, "b", bpawn_image),
       Piece(f"bp3", 4, 3, "b", bpawn_image),
       Piece(f"bp4", 5, 4, "b", bpawn_image),
       Piece(f"bp5", 6, 3, "b", bpawn_image),
       Piece(f"bp6", 7, 3, "b", bpawn_image),
       Piece(f"bp7", 8, 2, "b", bpawn_image),
       Piece(f"bq", 2, 7, "b", bqueen_image),
       Piece(f"bn1", 6, 2, "b", bpawn_image),
       Piece(f"br1", 8, 1, "b", brook_image),
       Piece(f"bb1", 6, 1, "b", bbishop_image)],
    False,
    False
   ),
   (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 7
       Piece(f"bk", 1, 5, "b", bking_image),
       [Piece(f"wk", 7, 8, "w", wking_image),
        Piece(f"wp1", 6, 7, "w", wpawn_image),
       Piece(f"wp2", 7, 7, "w", wpawn_image),
       Piece(f"wp3", 8, 6, "w", wpawn_image),
       Piece(f"wq", 2,5, "w", wqueen_image),
       Piece(f"wr1", 3, 8, "w", wrook_image),
       Piece(f"wn1", 4, 4, "w", wpawn_image),
       Piece(f"wn2", 6, 2, "w", wpawn_image),
    ],
    [  Piece(f"bk", 1, 5, "b", bking_image),
       Piece(f"bp1", 1, 2, "b", bpawn_image),
       Piece(f"bp2", 1, 3, "b", bpawn_image),
       Piece(f"bp3", 6, 3, "b", bpawn_image),
       Piece(f"bp4", 8, 2, "b", bpawn_image),
       Piece(f"br1", 1, 7, "b", brook_image),
   ],
    True,
    False
   ),
   (Piece(f"wk", 7, 6, "w", wking_image), #TEST CASE 8
       Piece(f"bk", 5, 1, "b", bking_image),
       [Piece(f"wk", 7, 6, "w", wking_image),
        Piece(f"wp1", 1, 6, "w", wpawn_image),
       Piece(f"wp2", 2, 7, "w", wpawn_image),
       Piece(f"wp3", 3, 7, "w", wpawn_image),
       Piece(f"wp4", 5, 6, "w", wpawn_image),
       Piece(f"wp5", 7, 7, "w", wpawn_image),
       Piece(f"wr1", 4, 7, "w", wrook_image),
       Piece(f"wb1", 2, 6, "w", wbishop_image)
    ],
    [  Piece(f"bk", 1, 4, "b", bking_image),
       Piece(f"bp1", 1, 2, "b", bpawn_image),
       Piece(f"bp2", 2, 2, "b", bpawn_image),
       Piece(f"bp3", 3, 2, "b", bpawn_image),
       Piece(f"bp4", 5, 4, "b", bpawn_image),
       Piece(f"bp5", 8, 3, "b", bpawn_image),
       Piece(f"bp6", 8, 5, "b", bpawn_image),
       Piece(f"br1", 5, 2, "b", brook_image),
```

```
        Piece(f"bn1", 3, 3, "b", bpawn_image),
        Piece(f"bb1", 5, 5, "b", bbishop_image)],
    True,
    True
    ),
    (Piece(f"wk", 3, 8, "w", wking_image), #TEST CASE 9
        Piece(f"bk", 4, 1, "b", bking_image),
        [Piece(f"wk", 3, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 2, 7, "w", wpawn_image),
        Piece(f"wp3", 3, 7, "w", wpawn_image),
        Piece(f"wq", 8, 2, "w", wqueen_image),
        Piece(f"wr1", 1, 1, "w", wrook_image),
        Piece(f"wb1", 6, 7, "w", wbishop_image)
    ],
    [   Piece(f"bk", 4, 1, "b", bking_image),
        Piece(f"bp1", 2, 5, "b", bpawn_image),
        Piece(f"bp2", 3, 5, "b", bpawn_image)],
    True,
    False
    ),
    (Piece(f"wk", 5, 8, "w", wking_image), #TEST CASE 10
        Piece(f"bk", 6, 2, "b", bking_image),
        [Piece(f"wk", 5, 8, "w", wking_image),
         Piece(f"wp1", 6, 6, "w", wpawn_image),
        Piece(f"wp2", 7, 5, "w", wpawn_image),
        Piece(f"wp3", 8, 7, "w", wpawn_image),
        Piece(f"wb1", 6, 8, "w", wbishop_image),
        Piece(f"wr1", 8, 8, "w", wrook_image),
        Piece(f"wn1", 3, 3, "w", wpawn_image),
    ],
    [   Piece(f"bk", 6, 2, "b", bking_image),
        Piece(f"bp1", 1, 2, "b", bpawn_image),
        Piece(f"bp2", 7, 2, "b", bpawn_image),
        Piece(f"bp3", 8, 2, "b", bpawn_image),
        Piece(f"bb1", 1, 8, "b", bbishop_image),
        Piece(f"bb2", 6, 4, "b", bbishop_image),
        Piece(f"br1", 1, 7, "b", brook_image),
        Piece(f"br2", 2, 8, "b", brook_image),
        Piece(f"bn1", 6, 3, "b", bpawn_image)],
    False,
    True
    ),
    (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 11
        Piece(f"bk", 5, 1, "b", bking_image),
        [Piece(f"wk", 7, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 3, 5, "w", wpawn_image),
        Piece(f"wp3", 6, 7, "w", wpawn_image),
        Piece(f"wp4", 7, 6, "w", wpawn_image),
        Piece(f"wp5", 8, 5, "w", wpawn_image),
        Piece(f"wq", 1, 5, "w", wqueen_image),
        Piece(f"wr1", 2, 8, "w", wrook_image),
        Piece(f"wr2", 6, 8, "w", wrook_image),
        Piece(f"wn1", 3, 2, "w", wpawn_image),
        Piece(f"wb1", 6, 3, "w", wbishop_image),
        Piece(f"wb2", 5, 7, "w", wpawn_image),
```

```
    ],
    [  Piece(f"bk", 5, 1, "b", bking_image),
       Piece(f"bp1", 1, 2, "b", bpawn_image),
       Piece(f"bp2", 4, 6, "b", bpawn_image),
       Piece(f"bp3", 6, 2, "b", bpawn_image),
       Piece(f"bp4", 8, 4, "b", bpawn_image),
       Piece(f"bq", 2, 3, "b", bqueen_image),
       Piece(f"br1", 1, 1, "b", brook_image),
       Piece(f"bn1", 2, 1, "b", bpawn_image),
       Piece(f"bb1", 6, 1, "b", bbishop_image),
       Piece(f"bb2", 5, 3, "b", bbishop_image)],
    False,
    True
    ),
   (Piece(f"wk", 6, 7, "w", wking_image), #TEST CASE 12
       Piece(f"bk", 8, 1, "b", bking_image),
       [Piece(f"wk", 6, 7, "w", wking_image),
       Piece(f"wr1", 8, 8, "w", wrook_image),
       Piece(f"wn1", 6, 3, "w", wpawn_image),
       Piece(f"wb1", 6, 1, "w", wbishop_image)
    ],
    [  Piece(f"bk", 8, 1, "b", bking_image)],
    False,
    True
    ),
   (Piece(f"wk", 5, 8, "w", wking_image), #TEST CASE 13
       Piece(f"bk", 5, 1, "b", bking_image),
       [Piece(f"wk", 5, 8, "w", wking_image),
        Piece(f"wp1", 1, 7, "w", wpawn_image),
       Piece(f"wp2", 2, 7, "w", wpawn_image),
       Piece(f"wp3", 3, 5, "w", wpawn_image),
       Piece(f"wp4", 6, 7, "w", wpawn_image),
       Piece(f"wp5", 7, 7, "w", wpawn_image),
       Piece(f"wp6", 8, 7, "w", wpawn_image),
       Piece(f"wq", 4, 1, "w", wqueen_image),
       Piece(f"wn1", 2, 8, "w", wknight_image),
       Piece(f"wb1", 3, 8, "w", wbishop_image),
       Piece(f"wn2", 6, 6, "w", wknight_image),
       Piece(f"wr1", 1, 8, "w", wrook_image),
       Piece(f"wn1", 8, 8, "w", wpawn_image),
    ],
    [  Piece(f"bk", 5, 1, "b", bking_image),
       Piece(f"bp1", 1, 2, "b", bpawn_image),
       Piece(f"bp2", 1, 3, "b", bpawn_image),
       Piece(f"bp3", 5, 4, "b", bpawn_image),
       Piece(f"bp4", 6, 2, "b", bpawn_image),
       Piece(f"bp5", 7, 2, "b", bpawn_image),
       Piece(f"bp6", 8, 2, "b", bpawn_image),
       Piece(f"br1", 1, 1, "b", brook_image),
       Piece(f"br2", 8, 1, "b", brook_image),
       Piece(f"bn1", 6, 3, "b", bpawn_image),
       Piece(f"bb1", 3, 1, "b", bbishop_image),
       Piece(f"bb2", 6, 1, "b", bbishop_image)],
    False,
    False
    ),
   (Piece(f"wk", 3, 2, "w", wking_image), #TEST CASE 14
```

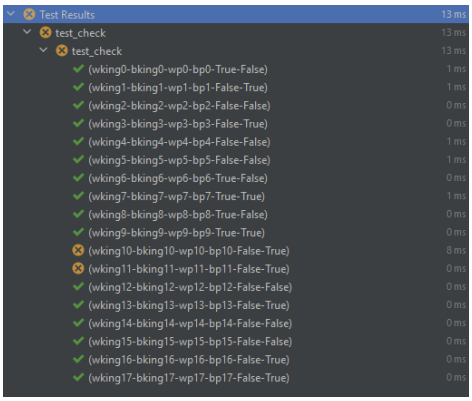
```
        Piece(f"bk", 7, 3, "b", bking_image),
        [Piece(f"wk", 3, 2, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 3, 6, "w", wpawn_image),
        Piece(f"wp3", 6, 7, "w", wpawn_image),
        Piece(f"wp4", 7, 6, "w", wpawn_image),
        Piece(f"wp5", 8, 7, "w", wpawn_image),
        Piece(f"wn1", 2, 5, "w", wknight_image),
        Piece(f"wr1", 1, 8, "w", wrook_image),
        Piece(f"wr2", 3, 2, "w", wrook_image),
    ],
    [  Piece(f"bk", 7, 3, "b", bking_image),
        Piece(f"bp1", 1, 3, "b", bpawn_image),
        Piece(f"bp2", 3, 5, "b", bpawn_image),
        Piece(f"bp3", 6, 3, "b", bpawn_image),
        Piece(f"bp4", 8, 3, "b", bpawn_image),
        Piece(f"br1", 4, 1, "b", brook_image),
        Piece(f"br2", 4, 8, "b", brook_image),
        Piece(f"bb1", 2, 2, "b", bbishop_image)],
    False,
    True
    ),
    (Piece(f"wk", 5, 8, "w", wking_image), #TEST CASE 15
        Piece(f"bk", 8, 1, "b", bking_image),
        [Piece(f"wk", 5, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 3, 6, "w", wpawn_image),
        Piece(f"wp3", 4, 5, "w", wpawn_image),
        Piece(f"wp4", 6, 7, "w", wpawn_image),
        Piece(f"wp5", 7, 7, "w", wpawn_image),
        Piece(f"wp6", 8, 7, "w", wpawn_image),
        Piece(f"wq", 5, 4, "w", wqueen_image),
        Piece(f"wn1", 7, 4, "w", wknight_image),
        Piece(f"wb1", 6, 2, "w", wbishop_image),
        Piece(f"wb2", 3, 8, "w", wbishop_image),
        Piece(f"wr1", 1, 8, "w", wrook_image),
        Piece(f"wn1", 8, 8, "w", wpawn_image),
    ],
    [  Piece(f"bk", 8, 1, "b", bking_image),
        Piece(f"bp1", 1, 2, "b", bpawn_image),
        Piece(f"bp2", 2, 2, "b", bpawn_image),
        Piece(f"bp3", 3, 2, "b", bpawn_image),
        Piece(f"bp4", 4, 2, "b", bpawn_image),
        Piece(f"bp5", 7, 3, "b", bpawn_image),
        Piece(f"bp6", 8, 2, "b", bpawn_image),
        Piece(f"bq", 4, 1, "b", bqueen_image),
        Piece(f"br1", 1, 1, "b", brook_image),
        Piece(f"br2", 6, 1, "b", brook_image),
        Piece(f"bn1", 1, 4, "b", bpawn_image),
        Piece(f"bb1", 3, 1, "b", bbishop_image)],
    False,
    False
    ),
    (Piece(f"wk", 3, 8, "w", wking_image), #TEST CASE 16
        Piece(f"bk", 5, 1, "b", bking_image),
        [Piece(f"wk", 3, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
```

```
        Piece(f"wp2", 2, 7, "w", wpawn_image),
        Piece(f"wp3", 3, 7, "w", wpawn_image),
        Piece(f"wp4", 5, 5, "w", wpawn_image),
        Piece(f"wp5", 7, 7, "w", wpawn_image),
        Piece(f"wq", 4, 7, "w", wqueen_image),
        Piece(f"wn1", 3, 6, "w", wknight_image),
        Piece(f"wb1", 2, 4, "w", wbishop_image),
        Piece(f"wb2", 3, 4, "w", wbishop_image),
        Piece(f"wr1", 4, 8, "w", wrook_image),
        Piece(f"wn1", 8, 4, "w", wpawn_image),
    ],
    [   Piece(f"bk", 5, 1, "b", bking_image),
        Piece(f"bp1", 1, 2, "b", bpawn_image),
        Piece(f"bp2", 2, 2, "b", bpawn_image),
        Piece(f"bp3", 5, 4, "b", bpawn_image),
        Piece(f"bp4", 6, 2, "b", bpawn_image),
        Piece(f"bp5", 8, 3, "b", bpawn_image),
        Piece(f"bq", 6, 6, "b", bqueen_image),
        Piece(f"br1", 1, 1, "b", brook_image),
        Piece(f"br2", 7, 5, "b", brook_image),
        Piece(f"bn1", 2, 3, "b", bpawn_image),
        Piece(f"bb1", 3, 1, "b", bbishop_image)],
    False,
    False
    ),
    (Piece(f"wk", 5, 8, "w", wking_image), #TEST CASE 15
        Piece(f"bk", 7, 1, "b", bking_image),
        [Piece(f"wk", 5, 8, "w", wking_image),
         Piece(f"wp1", 1, 7, "w", wpawn_image),
        Piece(f"wp2", 2, 7, "w", wpawn_image),
        Piece(f"wp3", 3, 6, "w", wpawn_image),
        Piece(f"wp4", 4, 5, "w", wpawn_image),
        Piece(f"wp5", 6, 7, "w", wpawn_image),
        Piece(f"wp6", 7, 5, "w", wpawn_image),
        Piece(f"wp7", 8, 5, "w", wpawn_image),
        Piece(f"wq", 5, 6, "w", wqueen_image),
        Piece(f"wn1", 4, 7, "w", wknight_image),
        Piece(f"wn2", 6, 6, "w", wknight_image),
        Piece(f"wb1", 4, 6, "w", wbishop_image),
        Piece(f"wr1", 4, 8, "w", wrook_image),
        Piece(f"wr2", 8, 8, "w", wrook_image),
    ],
    [   Piece(f"bk", 7, 1, "b", bking_image),
        Piece(f"bp1", 1, 3, "b", bpawn_image),
        Piece(f"bp2", 2, 4, "b", bpawn_image),
        Piece(f"bp3", 3, 2, "b", bpawn_image),
        Piece(f"bp4", 4, 4, "b", bpawn_image),
        Piece(f"bp5", 5, 3, "b", bpawn_image),
        Piece(f"bp6", 6, 2, "b", bpawn_image),
        Piece(f"bp7", 7, 2, "b", bpawn_image),
        Piece(f"bp8", 8, 3, "b", bpawn_image),
        Piece(f"bq", 4, 1, "b", bqueen_image),
        Piece(f"br1", 1, 1, "b", brook_image),
        Piece(f"br2", 6, 1, "b", brook_image),
        Piece(f"bn1", 3, 3, "b", bknight_image),
        Piece(f"bn2", 7, 7, "b", bknight_image),
        Piece(f"bb1", 3, 1, "b", bbishop_image)],
```

```
     False,
     True
     ),
    (Piece(f"wk", 7, 8, "w", wking_image), #TEST CASE 15
        Piece(f"bk", 7, 1, "b", bking_image),
        [Piece(f"wk", 7, 8, "w", wking_image),
         Piece(f"wp1", 1, 6, "w", wpawn_image),
        Piece(f"wp2", 2, 7, "w", wpawn_image),
        Piece(f"wp3", 3, 6, "w", wpawn_image),
        Piece(f"wp4", 5, 4, "w", wpawn_image),
        Piece(f"wp5", 6, 7, "w", wpawn_image),
        Piece(f"wq", 3, 2, "w", wqueen_image),
        Piece(f"wr1", 5, 8, "w", wrook_image),
        Piece(f"wr2", 6, 8, "w", wrook_image),
    ],
     [  Piece(f"bk", 7, 1, "b", bking_image),
        Piece(f"bp1", 1, 4, "b", bpawn_image),
        Piece(f"bp2", 2, 3, "b", bpawn_image),
        Piece(f"bp3", 4, 4, "b", bpawn_image),
        Piece(f"bp4", 5, 3, "b", bpawn_image),
        Piece(f"bp5", 7, 4, "b", bpawn_image),
        Piece(f"bq", 7, 5, "b", bqueen_image),
        Piece(f"br1", 6, 1, "b", brook_image),],
     False,
     True
     )



    ])


def test_check(wking,bking,wp,bp,expected_result,move):
    t = [[" " for i in range(8)] for i in range(8)]
    G = Game(wp, bp, move, wking, bking, [])


    if main.check_checker(G) is True:
        assert main.checkmate_checker(G) is expected_result
```

Above is the implementation of the test for checkmate detection, where the input is a random position for checkmate.

I will now use the test cases I planned above.



The first bug I realised from this was that I still have some issues with generality when passing in the attempted square for the piece moves that I am simulating. This should be quite an easy fix as I just need to change the square according to whether the check is up or down the board (relative to the direction of the check).

I noticed this from seeing which tests above failed. When I was manually testing my function I was biased towards checkmating as white (and hence up the board most of the time), luckily my tests made it clear I had not properly generalised for either way.

The next bug I noticed was that because I am simulating capturing moves, I need to re-add the attempted capture if the move is not valid. Below is the code with both of these changes made.

```
for checker in checking_pieces:
        if checker.name[1] == "b":
            for i in range(0, abs(king.xpos - checker.xpos) + 1):
                for piece in pieces:
                    tempx, tempy = piece.xpos, piece.ypos
                    piece.xpos, piece.ypos = checker.xpos + (numpy.sign(king.xpos - checker.xpos) * i),
                                                                    checker.ypos + (
                                    numpy.sign(king.ypos - checker.ypos) * i)
                    if i == 0:
                        G.remove(checker)
                        takenpiece = checker
                        if checker.name[0] == "w":
                            wp.remove(checker)
                        else:
                            bp.remove(checker)
                        tempitem = checker
                    if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                                  tempy) and not check_checker(G,wp, bp, wking, bking):
                        checkmate = False
                        piece.xpos = tempx
                        piece.ypos = tempy
                    else:
                        piece.xpos = tempx
                        piece.ypos = tempy
                    if tempitem:
                        G.ap.append(tempitem)
                        if tempitem.name[0] == "w":
                            wp.append(tempitem)
                        else:
                            bp.append(tempitem)
                        tempitem = None
        elif checker.name[1] == "r":
            if checker.xpos - king.xpos == 0:
                for i in range(0, abs(king.ypos - checker.ypos)):
                    for piece in pieces:
                        tempx = piece.xpos
                        tempy = piece.ypos
                        piece.ypos = checker.ypos + (numpy.sign(king.ypos - checker.ypos) * i)
                        if i == 0:
                            G.ap.remove(checker)
                            takenpiece = checker
                            if checker.name[0] == "w":
                                wp.remove(checker)
                            else:
                                bp.remove(checker)
                            tempitem = checker
                        if move_valid(G,piece, checker.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                                      tempy) and not check_checker(G,wp, bp, wking, bking):
                            checkmate = False
                            piece.xpos = tempx
                            piece.ypos = tempy
                        else:
                            piece.xpos = tempx
                            piece.ypos = tempy
                        if tempitem:
                            G.ap.append(tempitem)
                            if tempitem.name[0] == "w":
                                wp.append(tempitem)
                            else:
                                bp.append(tempitem)
                            tempitem = None
            if checker.ypos - king.ypos == 0:
                for i in range(0, abs(wking.xpos - checker.xpos) + 1):
                    for piece in pieces:
                        tempx = piece.xpos
                        tempy = piece.ypos
                        piece.xpos = checker.xpos + (numpy.sign(king.xpos - checker.xpos) * i)
                        piece.ypos = checker.ypos
```

```
                    if i == 0:
                        G.ap.remove(checker)
                        takenpiece = checker
                        if checker.name[0] == "w":
                            wp.remove(checker)
                        else:
                            bp.remove(checker)
                        tempitem = checker
                    if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx, tempy):
                        checkmate = False
                        piece.xpos = tempx
                        piece.ypos = tempy
                    else:
                        piece.xpos = tempx
                        piece.ypos = tempy
                    if tempitem:
                        G.ap.append(tempitem)
                        if tempitem.name[0] == "w":
                            wp.append(tempitem)
                        else:
                            bp.append(tempitem)
                        tempitem = None
        pass
    elif checker.name[1] == "q":
        if checker.xpos - king.xpos == 0:
            for i in range(0, abs(king.ypos - checker.ypos) + 1):
                for piece in pieces:
                    tempx = piece.xpos
                    tempy = piece.ypos
                    piece.xpos = checker.xpos
                    piece.ypos = checker.ypos + (constant * i)
                    if i == 0:
                        G.ap.remove(checker)
                        takenpiece = checker
                        if checker.name[0] == "w":
                            wp.remove(checker)
                        else:
                            bp.remove(checker)
                        tempitem = checker

                    if move_valid(G,piece, checker.xpos, checker.ypos + (constant * i), wp, bp, wking,
                                                                                        bking, tempx,
                                  tempy):
                        checkmate = False
                        piece.xpos = tempx
                        piece.ypos = tempy
                    else:
                        piece.xpos = tempx
                        piece.ypos = tempy
                    if tempitem:
                        G.ap.append(tempitem)
                        if tempitem.name[0] == "w":
                            wp.append(tempitem)
                        else:
                            bp.append(tempitem)
                        tempitem = None
        if checker.ypos - king.ypos == 0:
            for i in range(0, abs(wking.xpos - checker.xpos) + 1):
                for piece in pieces:
                    tempx = piece.xpos
                    tempy = piece.ypos
                    piece.xpos = checker.xpos + (numpy.sign(king.xpos - checker.xpos) * i)
                    piece.ypos = checker.ypos
```

```
            if i == 0:
                G.ap.remove(checker)
                takenpiece = checker
                if checker.name[0] == "w":
                    wp.remove(checker)
                else:
                    bp.remove(checker)
                tempitem = checker
            if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                                    tempy):
                checkmate = False
                piece.xpos = tempx
                piece.ypos = tempy
            else:
                piece.xpos = tempx
                piece.ypos = tempy
            if tempitem:
                G.ap.append(tempitem)
                if tempitem.name[0] == "w":
                    wp.append(tempitem)
                else:
                    bp.append(tempitem)
                tempitem = None
    else:
        for i in range(0, abs(king.xpos - checker.xpos) + 1):
            for piece in pieces:
                tempx = piece.xpos
                tempy = piece.ypos
                piece.xpos = checker.xpos + (numpy.sign(king.xpos - checker.xpos) * i)
                piece.ypos = checker.ypos + (numpy.sign(king.ypos - checker.ypos) * i)
                if i == 0:
                    G.ap.remove(checker)
                    takenpiece = checker
                    if checker.name[0] == "w":
                        wp.remove(checker)
                    else:
                        bp.remove(checker)
                    tempitem = checker
                if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                              tempy):
                    checkmate = False
                    piece.xpos = tempx
                    piece.ypos = tempy
                else:
                    piece.xpos = tempx
                    piece.ypos = tempy
                if tempitem:
                    G.ap.append(tempitem)
                    if tempitem.name[0] == "w":
                        wp.append(tempitem)
                    else:
                        bp.append(tempitem)
                    tempitem = None
```

In this function, I noticed that I have a block of code that is repeated multiple times. This block involves reverting a piece's position from a simulated move back to its original position. To reduce repetition in future iterations, I plan to refactor this code into a separate function.

I have just completed the implementation of the final section of the code that handles checkmate when the queen is checking diagonally. This portion of the code is largely similar to the checkmate detection for bishops. In the next step, I

will add checkmate detection for pawns and knights, which should be relatively straightforward because these pieces do not create a line of attack like bishops and queens do. The only way to escape checkmate in these cases is to move the king or capture the attacking piece.

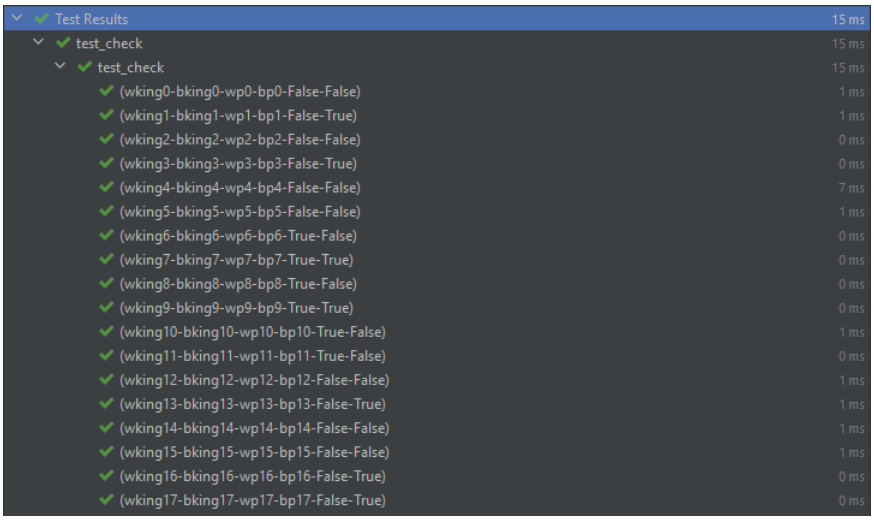
```
elif checker.name[1] == "n":
            for piece in pieces:
                if piece.name[1] == "k":
                    continue
                tempx = piece.xpos
                tempy = piece.ypos
                piece.xpos = checker.xpos
                piece.ypos = checker.ypos
                G.ap.remove(checker)
                takenpiece = checker
                if checker.name[0] == "w":
                    wp.remove(checker)
                else:
                    bp.remove(checker)
                tempitem = checker
                if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                              tempy):
                    checkmate = False
                    piece.xpos = tempx
                    piece.ypos = tempy
                else:
                    piece.xpos = tempx
                    piece.ypos = tempy
                if tempitem:
                    G.ap.append(tempitem)
                    if tempitem.name[0] == "w":
                        wp.append(tempitem)
                    else:
                        bp.append(tempitem)
                    tempitem = None
```

```
        elif checker.name[1] == "p":
            for piece in pieces:
                tempx = piece.xpos
                tempy = piece.ypos
                piece.xpos = checker.xpos
                piece.ypos = checker.ypos
                G.ap.remove(checker)
                takenpiece = checker
                if checker.name[0] == "w":
                    wp.remove(checker)
                else:
                    bp.remove(checker)
                tempitem = checker
                if move_valid(G,piece, piece.xpos, piece.ypos, wp, bp, wking, bking, tempx,
                              tempy):
                    checkmate = False
                    piece.xpos = tempx
                    piece.ypos = tempy
                else:
                    piece.xpos = tempx
                    piece.ypos = tempy
                if tempitem:
                    G.ap.append(tempitem)
                    if tempitem.name[0] == "w":
                        wp.append(tempitem)
                    else:
                        bp.append(tempitem)
                    tempitem = None
```

The last thing I need to do to complete **checkmate_checker()** is to add in detection for getting out of check my moving the king. Again, this should be a straightforward addition as there are only ever 8 vectors a king can move so I simply need to check if any of these are valid.

```
    for vector in king_vectors:  # checks if there is anywhere the king can legally move to
        king.xpos, king.ypos = king.xpos + vector[0], king.ypos + vector[1]
        blockage = False
        for piece in pieces:
            if (piece.xpos, piece.ypos) == (king.xpos, king.ypos) and piece != king:
                blockage = True  # blockage sees if there is a piece of the same colour at the new square
        if move_valid(G,king, king.xpos, king.ypos, wp, bp, wking, bking, tempx, tempy) and not check_checker
                                                                (G,wp, bp,wking,bking) and king.xpos > 0 and
                                                                king.ypos > 0 and not blockage:
            checkmate = False
            continue
        king.xpos = tempx
        king.ypos = tempy
```

Upon testing, this worked straightaway as can be seen below.



3.1.3 Prototype(s)

Below is the final prototype from Development Cycle 1:

Prototype 1

In the video below I demonstrate a Checkmate taking place. Although it is not clear from the pygame window, "checkmate" is printed in the console and no moves can be made

Prototype 1 Part 2

3.1.4 Review

Objects Achieved:

- The graphics of the board and pieces have been implemented
- The legal movements of pieces have been implemented
- The program only lets the correct player move
- Checkmate detection has been implemented

Now I will present the current prototype to my stakeholder for review and incorporate their feedback as I plan the next steps in the second development cycle.

My stakeholder was pleased with the progress of the prototype and provided the following feedback:

- The stakeholder liked the graphics and smooth movement of the pieces
- Most of his expectations for the base game of chess were met

- He suggested adding sounds to assist beginners, like himself
- He noticed a small issue with the graphics where the black piece is always displayed underneath the white piece
- He recommended adding an indication of the winner when the game ends
- He emphasized the importance of including clocks in the final iteration to keep play fast

I found my stakeholder's feedback to be very helpful. In the next development cycle, I plan to focus on adding the "Bughouse" aspect to my project. Whilst I do plan to incorporate the smaller improvements suggested by Zack, I will likely address them in Development Cycle 3 after I have implemented the "Bughouse" aspect.

3.2 Development Cycle 2

3.2.1 Aims

In this development cycle I aim to design the interaction between boards on separate clients, with moves being processed by the server. After this I will upgrade to four boards with two games running simultaneously. Finally I will need to add implementation of the bughouse rules, meaning that captured pieces will be passed to the other board with the option to be placed.

3.2.2 Development

To begin the development I first focused on making the connection between a single client and server using socket. To test this, I simply sent the co-ordinates of a game from the server to the client and had the client display it.

```
# CLIENT
import pygame
import socket
import pickle

HOST = "127.0.0.1"  # The server's hostname or IP address
PORT = 65432  # The port used by the server


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    connection, address = s.accept()
    with connection:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            received= pickle.loads(data)
pygame.init()
```

```
screen = pygame.display.set_mode((1000, 800))
clock = pygame.time.Clock()

#Class defintions, image loading and piece creation etc...

for count,piece in enumerate(ap): #update the pieces to the
                                  positions received
    piece.update(received[count][0],received[count][1])

while True:
    screen.fill(white)
    screen.blit(board_image, (200, 75))
    for i in ap: #continuously display the pieces
        screen.blit(i.image, (i.placerx, i.placery))
    pygame.display.update()
```

```
#SERVER
import socket
import pickle

HOST = "127.0.0.1"  # Standard loopback interface address (
                                  localhost)
PORT = 65432  # Port to listen on (non-privileged ports are >
                                  1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    plain_data = #list of lists of piece co-ords
    data = pickle.dumps(plain_data)
    s.send(data)
    data = s.recv(1024)
    data = pickle.loads(data)
```

The code above is merely a proof of concept to myself that I can exchange piece positions and display pieces on a client. To help me with this I used https://realpython.com/python-sockets/. Now I will take the next step to have the client send move requests and the server process the moves and send them back. If it goes to plan, it should play the same as the prototype at the end of Development Cycle 1 but a server will be performing the calculations.

Using the code from Development Cycle 1 and the flowchart from my design section, implementing server processing should be quite straightforward.

```
#SERVER

#Functions, imports and classes from Development Cycle 1

def process_request(request):
    for i in G.ap:
        if i.name == request[0]:
            item = i
            xsquare = request[1]
            ysquare = request[2]
            newposx = item.xpos
            newposy = item.ypos
```

```
                break

        global takenpiece
        global turn
        takenpiece = None
        tempitem = None


        tempx, tempy = item.xpos, item.ypos
        item.xpos, item.ypos = xsquare, ysquare
        for i in G.ap:
            if i.xpos == xsquare and i.ypos == ysquare and i.name[0
                                                ] != item.name[0]:
                G.ap.remove(i)
                takenpiece = i
                if i.name[0] == "w":
                    G.wp.remove(i)
                else:
                    G.bp.remove(i)
                tempitem = i

        if item.name[0] == "w" and G.move:
            turn = True
        elif item.name[0] == "b" and not G.move:
            turn = True
        else:
            turn = False



        movevalid = move_valid(G,item, xsquare, ysquare, newposx,
                                           newposy)

        if movevalid and turn:
            tempitem = None
        if not movevalid or not turn:
            item.placerx = 125 + tempx * 75
            item.xpos = tempx
            item.placery = tempy * 75
            item.ypos = tempy
            if tempitem:
                G.ap.append(tempitem)
                if tempitem.name[0] == "w":
                    G.wp.append(tempitem)
                else:
                    G.bp.append(tempitem)
            return False
        if movevalid and turn:
            G.move = not G.move
            return True
```

As you can see from the code above, I decided to update the Game object on the server this means both the server and the client will have an up-to-date Game object so the server can correctly validate any moves.

In a future iteration I could slim down the data on the server as it does not need

to know the placerx and placery positions, as it never has to place pieces. By the same logic it also doesn't need to know or load the relevant piece images.

```
pygame.init()
screen = pygame.display.set_mode((1000, 800))
clock = pygame.time.Clock()

#Load images, define Classes, create pieces, define Snapper(x,y
                                    ) etc

 while True:
    clock.tick(120)
    screen.fill(white)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
    screen.blit(board_image, (200, 75)) #display board
    for i in G.ap:
        screen.blit(i.image, (i.placerx, i.placery)) #display
                                                pieces
        if event.type == pygame.MOUSEBUTTONDOWN and event.
                                                button == 1:
            x, y = pygame.mouse.get_pos()
            for j in range(1, 9):
                if 200 + (j - 1) * 75 < x < 200 + j * 75:
                    newposx = j
                    break
            for j in range(1, 9):
                if 75 + (j - 1) * 75 < y < 75 + j * 75:
                    newposy = j
                    break
            for i in G.ap:
                if i.xpos == newposx and i.ypos == newposy:
                    i.placerx = x
                    i.placery = y
                    item = i
        if pygame.mouse.get_pressed()[0] and item:
            x, y = pygame.mouse.get_pos()
            item.placerx = x - 75 / 2
            item.placery = y - 75 / 2
        if not pygame.mouse.get_pressed()[0]:

            try:
                x, y = pygame.mouse.get_pos()
                xsquare, ysquare = snapper(x,
                                           y)
                        # xsquare and ysquare
                        #are the squares the piece is
                        #trying to be placed on
                item.placerx = 125 + xsquare * 75
                item.placery = ysquare * 75
                movevalid = True
                tempx = item.xpos
                tempy = item.ypos
                item.xpos = xsquare
                item.ypos = ysquare
```

```
                    for i in G.ap:
                        if i.xpos == xsquare and i.ypos == ysquare
                                                            and i.
                                                            name[0] !
                                                            = item.
                                                            name[0]:
                            G.ap.remove(i)
                            takenpiece = i
                            if i.name[0] == "w":
                                G.wp.remove(i)
                            else:
                                G.bp.remove(i)
                            tempitem = i

                    data = pickle.dumps([item.name, xsquare,
                                                        ysquare])
                    client_socket.sendall(data)
                    result = pickle.loads(client_socket.recv(1024))
                    if result:
                        tempitem = None
                    if not result: #revert to original position if
                                                        illegal move
                        item.placerx = 125 + tempx * 75
                        item.xpos = tempx
                        item.placery = tempy * 75
                        item.ypos = tempy
                        if tempitem:
                            G.ap.append(tempitem)
                            if tempitem.name[0] == "w":
                                G.wp.append(tempitem)
                            else:
                                G.bp.append(tempitem)
                    if result: #if the move is valid, change whose
                                                        move it is
                        G.move = not G.move

                except AttributeError: #excepting if no piece has
                                                    been picked up
                    pass
                except NameError:
                    pass
                item = None

    pygame.display.update()
```

Helped by the large amount of code and design I had already done, this step was straightforward as expected. The next step to take is upgrade this to two clients, each controlling one colour.

There aren't many changes to be made on the server side to reach this. The main thing to do is loop between listening to each client, as well as sending updates to **client 1** if **client 2** has made a valid move and vice versa. This **algorithm** was described in the analysis section.

```
    #unchanged code
```

```
 server_socket = socket.socket(socket.AF_INET, socket.
                                    SOCK_STREAM) #create a
                                    socket object
server_socket.bind(('localhost', 8000)) #binds socket to local
                                   port
server_socket.listen(2) #listening for 2 connections


client1, address1 = server_socket.accept() #accepts a
                                   connection from each client
print(f'Connected to {address1}')
client2, address2 = server_socket.accept()
print(f'Connected to {address2}')



while True:
    for client in (client1, client2): #alternates between
                                       requests from client1 and
                                        client2 as it can't deal
                                        with both simultaneously
        client.settimeout(0.00001) #sets a short timeout for
                                           each client and tries
                                            to recieve data
        try:
            data = client.recv(1024)
        except socket.timeout: #if there is a timeout error, it
                                           continues to next
                                           client
            continue
        if not data:
            break
        request = pickle.loads(data)
        result = process_request(request)
        client.sendall(pickle.dumps(result))
        if result:
            #if a move is valid it send to the other client so
                                              their board can
                                              update
            if client == client1:
                while True:
                    try:
                        client2.sendall(data)
                        #keep trying to send until recieved
                        break
                    except OSError:
                        pass
            else:
                while True:
                    try:
                        client1.sendall(data)
                        break
                    except OSError:
                        pass
```

On the client side the sending of moves can remain the same. The main change is

it also needs to listen to messages from the server informing it that the opponent has made a valid move, and update its pieces accordingly. Below I will show the player with the white pieces.

```
#unchanged code

clock.tick(120)
screen.fill(white)

client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 8000))



while True:

    #unchanged code for sending and dealing with result to/from
                                    server

                if G.move == True: #only send if it is your move as
                                                    otherwise always
                                                    invalid
                #for player with black pieces change this to if G.
                                                   move == False
                    data = pickle.dumps([item.name, xsquare,
                                                      ysquare])
                    while True:
                        try:
                            client_socket.sendall(data)
                            break
                        except OSError:
                            pass
                    client_socket.settimeout(100000)
                    result = pickle.loads(client_socket.recv(1024))
                else:
                    result = False

                if result:
                    tempitem = None
                if not result:
                    item.placerx = 125 + tempx * 75
                    item.xpos = tempx
                    item.placery = tempy * 75
                    item.ypos = tempy
                    if tempitem:
                        G.ap.append(tempitem)
                        if tempitem.name[0] == "w":
                            G.wp.append(tempitem)
                        else:
                            G.bp.append(tempitem)
                if result:
                    G.move = not G.move

    #unchanged code for sending and dealing with result to/from
                                    server
            except AttributeError:
```

```
            pass
        except NameError:
            pass
        item = None

        client_socket.settimeout(0.0000001)  # Set a short
                                             timeout
        try:
            G.move = not G.move #change whose move it is as
                                              server will only
                                              send valid moves
            for i in G.ap:
                if i.xpos == result[1] and i.ypos == result[2]:
                    #delete captured pieces
                    G.ap.remove(i)
                    if i.name[0] == "w":
                        G.wp.remove(i)
                    else:
                        G.bp.remove(i)
                if i.name == result[0]: #update moved pieces
                    i.xpos, i.ypos = result[1],result[2]
                    i.placerx = 125 + result[1] * 75
                    i.placery = result[2] * 75
        except socket.timeout:
            # No result was received within the timeout
            continue
pygame.display.update()
```

After this I had a working game between two clients with server processing. However, aside from not having four players or bughouse rules, there are still a few things missing. I will probably deal with the second item before implementing the bughouse rules as this will most likely make it easier for ensuring the captured pieces go to the right place.

- Ending the game
- The player with black should have the board from black's perspective

I plan to handle the first point once I have implemented the bughouse rules. I think I will tackle the second before I have implemented the bughouse rules, to make sure all the new pieces go where they are supposed to. I don't expect the change to be too difficult as it is just a case of displaying the board with the pieces "mirrored" as well as mirroring the numbered co-ordinates when given pixel co-ordinates for black piece players.

As I have 4 clients, 2 playing white and 2 playing black, I think it would make sense to use some sort of signal so that **client1** is always paired with **client2** and **client3** is always paired with **client4**. In order to achieve this, I will make each client send their respective name when they connect, and then name them accordingly on the server side.

On the client side it will look like this:

```
client_socket.send("client1".encode())
```

And on the server side it will look like this:

```
    for client in range(4): #accept four clients
        client_socket, client_address = server_socket.accept()
        client_name = client_socket.recv(1024).decode()
        exec(client_name + " =  client_socket") #associate with the
                                            name sent
        print(f"Connected to {client_name}")
```

Now that I have done this, the next thing I need to do is update the boards for the people playing black, so that the board is from their perspective.

This was quite a simple change which essentially just involves placing the piece on the opposite squares for the black players but keeping their "position" (**xpos** and **ypos**) the same, this essentially means all previous functions can be kept the same. These changes are only present in the main game loop, below is the updated version in which I have included comments to show the changes:

```
while True:
    clock.tick(120)
    screen.fill(white)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False
    screen.blit(board_image, (200, 75))
    for i in G.ap:
        #Place the pieces on the opposite squares
        screen.blit(i.image, (9*75 - i.placerx + 250, 9*75 - i.
                                            placery))
        if event.type == pygame.MOUSEBUTTONDOWN and event.button ==
                                            1:
            x, y = pygame.mouse.get_pos()
            for j in range(1, 9):
                if 200 + (j - 1) * 75 < x < 200 + j * 75:
                    newposx = j
                    break
            for j in range(1, 9):
                if 75 + (j - 1) * 75 < y < 75 + j * 75:
                    newposy = j
                    break
            for i in G.ap:
                #seeing if a piece is where a player is clicking
                if i.xpos == 9-newposx and i.ypos == 9-newposy:
                    i.placerx = x
                    i.placery = y
                    item = i
        if pygame.mouse.get_pressed()[0] and item:
            x, y = pygame.mouse.get_pos()
            #updating the placer
            item.placerx = 9*75 - (x - 75 / 2) + 250
            item.placery = 9*75 - (y - 75 / 2)
        if not pygame.mouse.get_pressed()[0]:

            try:
                x, y = pygame.mouse.get_pos()
                xsquare, ysquare = snapper(x,
```

```
                                        y)
                #updating the placer
                item.placerx = 125 + (9-xsquare) * 75
                item.placery = (9-ysquare) * 75
                movevalid = True
                tempx = item.xpos
                tempy = item.ypos
                item.xpos = xsquare
                item.ypos = ysquare
                for i in G.ap:
                    #seeing if a piece has been captured
                    if i.xpos == 9-xsquare and i.ypos == 9 -
                                                    ysquare and i
                                                    .name[0] !=
                                                    item.name[0]:
                        G.ap.remove(i)
                        takenpiece = i
                        if i.name[0] == "w":
                            G.wp.remove(i)
                        else:
                            G.bp.remove(i)
                        tempitem = i


                if G.move == False:
                    #send the move to the server (with the regular
                                                    coordinates)
                    data = pickle.dumps([item.name, 9 - xsquare, 9
                                                    - ysquare])
                    item.xpos = 9 - item.xpos
                    item.ypos = 9 - item.ypos

#code continues
```

The next stage is to deal with passing captured pieces to the other board. For the moment, I will not focus on making sure the pieces go to the correct square, but I will deal with that later. As it isn't possible to send an image over socket I will simply send the name of the piece, and update the image once it is received according to the name. For this reason I added a method to the **Piece** class that will update a piece's image.

```
    def update_image(self):
        images = {
            "wp": wpawn_image,
            "wq": wqueen_image,
            "wr": wrook_image,
            "wb": wbishop_image,
            "wn": wknight_image,
            "wk": wking_image,
            "bp": bpawn_image,
            "bq": bqueen_image,
            "br": brook_image,
            "bb": bbishop_image,
            "bn": bknight_image,
            "bk": bking_image
        }
```

```
        self.image = images.get(self.name[:2],self.image)
```

Calling this method simply sets the pieces image to the correct one.

If a piece is captured I will simply send the name of the piece, combined with "new" (e.g "wqnew") simply to indicate it is a new piece. The reason for this will become clear shortly. As I will just send a string, unlike a regular move in which a list is sent, I can update the client to recognise when it has received a new piece:

```
    try:
        result = pickle.loads(client_socket.recv(1024))
        if type(result) == str:
            newpiece = Piece(result, 0, 8, result[0])
            newpiece.update_image()
            G.ap.append(newpiece)
            getattr(G,newpiece.colour+"p").append(newpiece)
            #this is a new way of appending a piece to the right
                                              list according to it'
                                              s colour neatly
        else:
            G.move = not G.move
            for i in G.ap:
                if i.xpos == result[1] and i.ypos == result[2]:
                    G.ap.remove(i)
                    if i.name[0] == "w":
                        G.wp.remove(i)
                    else:
                        G.bp.remove(i)
                if i.name == result[0]:
                    i.xpos, i.ypos = result[1],result[2]
                    i.placerx = 125 + result[1] * 75
                    i.placery = result[2] * 75
```

In the code above I point out a new way of appending a piece to the correct list depending on its colour. This is much neater than using if-else statements like I have been doing before. I won't worry about this now but hopefully I will update this in a future iteration (for my own conscience).

On the server-side I updated the **process_request** function to also return the name of a captured piece:

```
        for i in gamenum.ap:
            if i.xpos == xsquare and i.ypos == ysquare and i.name[0
                                                  ] != item.name[0]:
                captured = True

        #function continues

        if not movevalid or not turn:
            item.placerx = 125 + tempx * 75
            item.xpos = tempx
            item.placery = tempy * 75
            item.ypos = tempy
            if tempitem:
```

```
                gamenum.ap.append(tempitem)
                if tempitem.name[0] == "w":
                    gamenum.wp.append(tempitem)
                else:
                    gamenum.bp.append(tempitem)
            return False, None
        if movevalid and turn:
            gamenum.move = not gamenum.move
            try:
                #returns move validity as well as the captured
                                                    piece (if
                                                    applicable)
                return True, tempitem.name
            except AttributeError:
                #if there is no captured piece, just returns
                                                    validity and None
                return True, None
```

The main loop then deals with the return value of **process_request** like so:

```
while True:
    for client in (client1, client2): #alternates between requests from client1 and client2 as it can't
                                                                    deal with both simultaneously
        client.settimeout(0.00001)
        try:
            data = client.recv(1024)
        except socket.timeout:
            continue
        if not data:
            break
        request = pickle.loads(data)
        print(request)
        result, taken = process_request(request,G1)
        #assigns taken to taken piece (or None)
        client.sendall(pickle.dumps(result))
        if result:  # if a move is valid it send to the other client so their board can update
            if client == client1:
                while True:
                    try:
                        if taken:
                        #if taken updates the servers board with the new piece and sends the name to
                                                                                    other board
                            newpiece = pickle.dumps(taken+"new")
                            client3.sendall(newpiece)
                            client4.sendall(newpiece)
                            newpiece = Piece(taken+"new", 0, 8, taken[0])
                            G2.ap.append(newpiece)
                            getattr(G2, newpiece.colour + "p").append(newpiece)
                        client2.sendall(data)  # keep trying to send until recieved
                        break
                    except OSError:
                        pass
            else:
                while True:
                    try:
                        if taken:
                        #if taken updates the servers board with the new piece and sends the name to
                                                                                    other board
                            newpiece = pickle.dumps(taken+"new")
                            client3.sendall(newpiece)
                            client4.sendall(newpiece)
                            newpiece = Piece(taken+"new", 0, 8, taken[0])
                            G2.ap.append(newpiece)
                            getattr(G2, newpiece.colour + "p").append(newpiece)
                        client1.sendall(data)
                        break
                    except OSError:
                        pass

    #same changes for the other board
```

The next thing to do is to add a method to the **Piece** class to update the position, depending on the type of piece. This is because I want each type of captured piece to appear on a specific square on the side of the board. Note that multiple of the same type of piece will stay on one square but I will do something to show how many pieces are on one square later.

```
    def update_position(self):
        positions = {
            "wp": (0,8),
            "wq": (0,4),
            "wr": (0,5),
            "wb": (0,6),
            "wn": (0,7),
```

```
            "bp": (9,1),
            "bq": (9,5),
            "br": (9,4),
            "bb": (9,3),
            "bn": (9,2)
        }
        self.xpos, self.ypos = positions.get(self.name[:2])[0],
                                            positions.get(self.name[:
                                            2])[1]
        self.placerx = 125 + self.xpos * 75
        self.placery = self.ypos * 75
```

In accordance to this I will also add the line:

```
    newpiece.update_position()
```

after the line where I update the image. As well as this, I will also update the for loops that check to see if a player is trying to move a piece to include the squares on either side of the board:

```
    if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
        x, y = pygame.mouse.get_pos()
        for j in range(0, 10): #updated to let players pick up
                                            pieces on the side
            if 200 + (j - 1) * 75 < x < 200 + j * 75:
                newposx = j
                break
        for j in range(0, 10): #updated to let players pick up
                                            pieces on the side
            if 75 + (j - 1) * 75 < y < 75 + j * 75:
                newposy = j
                break
```

I also need to update the **move_valid** function on the server side to allow players to place a captured piece anywhere on the board where there is not already pieces. Pawns are also not allowed to be placed on the 8th rank.

I added in the following code towards the start:

```
    if newposx <= 0 or newposx >= 9:
        if (item.name[:1] == "wp" and ysquare == 8) or (item.name[:
                                            1] == "bp" and ysquare ==
                                             1) or piecethere(G,
                                            xsquare,ysquare,item):
            return False
        else:
                return True
```

This returns True unless a pawn is trying to be placed on the 8th rank or any piece is trying to be placed on a full square.

The last thing I need to do to complete the bughouse aspect is to add an indication of how many captured pieces are on a square. In order to do this

I think the best approach is to store images for the numbers 2-8 and display them on the screen when appropriate. I will create a "bubble" for all the squares where captured pieces can be and treat them as objects.

For the bubble images I simply created them using some python code which I will not explore here.

Loading the images:

```
number2 = pygame.image.load(r"Images\2.png")
number3 = pygame.image.load(r"Images\3.png")
number4 = pygame.image.load(r"Images\4.png")
number5 = pygame.image.load(r"Images\5.png")
number6 = pygame.image.load(r"Images\6.png")
number7 = pygame.image.load(r"Images\7.png")
number8 = pygame.image.load(r"Images\8.png")
```

Bubble class:

```
class Bubble:
    def __init__(self,xpos,ypos):
        self.xpos = xpos
        self.ypos = ypos
        self.counter = 0
        self.image = None

    def add(self,numb=1):
        self.counter += numb

    def update_image(self):
        images = {
            "0": None,
            "1": None,
            "2": number2,
            "3": number3,
            "4": number4,
            "5": number5,
            "6": number6,
            "7": number7,
            "8": number8,
        }
        self.image = images.get(str(self.counter))
```

Creating bubbles:

```
bubbles = []
for i in range(8,3,-1):
    bubbles.append(Bubble(0,i))

for i in range(1,6):
    bubbles.append(Bubble(9,i))
```

Now in the original block of code where clients deal with incoming pieces we will also add to a bubbles counter if a new piece is going to that square:

```
    if type(result) == str:
        newpiece = Piece(result, 0, 8, result[0])
        newpiece.update_image()
        newpiece.update_position()
        G.ap.append(newpiece)
        for bubble in bubbles:
            if bubble.xpos == newpiece.xpos and bubble.ypos ==
                                                newpiece.ypos:
                bubble.add(1)
                #add to bubble counter
        getattr(G.newpiece.colour+"p").append(newpiece)
```

Now the fact that the word "new" will come in use as if a "new" piece is moved, we can see if it was on any of the bubble squares and update them accordingly.

```
    if result:
        tempitem = None
        if item.name[-1] == "w":
            for bubble in bubbles:
                if bubble.xpos == tempx and bubble.ypos == tempy:
                    bubble.add(-1)
```

The very last thing to do for this development cycle is to update all the boards if checkmate happens on either board.

To do this I simply added in an if statement that makes use of the function previously made and tested.

```
    if checkmate_checker(gamenum):
        print("checkmate")
        return "checkmate", None
```

The first step, is to update the server to actually send "checkmate" back to the clients. To do this I just need to update the loop for both boards to send "checkmate" to all the clients if the result returned from **process_request** is "checkmate":

```
    if result == "checkmate":
        for client in (client1,client2,client3,client4):
            client.sendall(pickle.dumps("checkmate"))
```

The next important thing to do is to update the clients to be able to deal with being sent "checkmate". At this stage it is probably suitable to display some text to let all the users now that checkmate has been delivered. At a latter stage in development it would also be appropriate to show which board it has happened on and who has been checkmated.

In order to do this I need to update the part of the code that sends a move and receives a result, as well as the part which receives moves from the opponent/other board, as checkmate may come on any player's move. To do that I will add this code to each part:

```
    if result == "checkmate":
        #if checkmate then display
        print("CHECKMATE")
        screen.blit(font.render("CHECKMATE", True, black), (400,0))
        checkmate = True
```

I will also add this to the bottom of the main game loop so that the board remains on the screen, but moves are not playable:

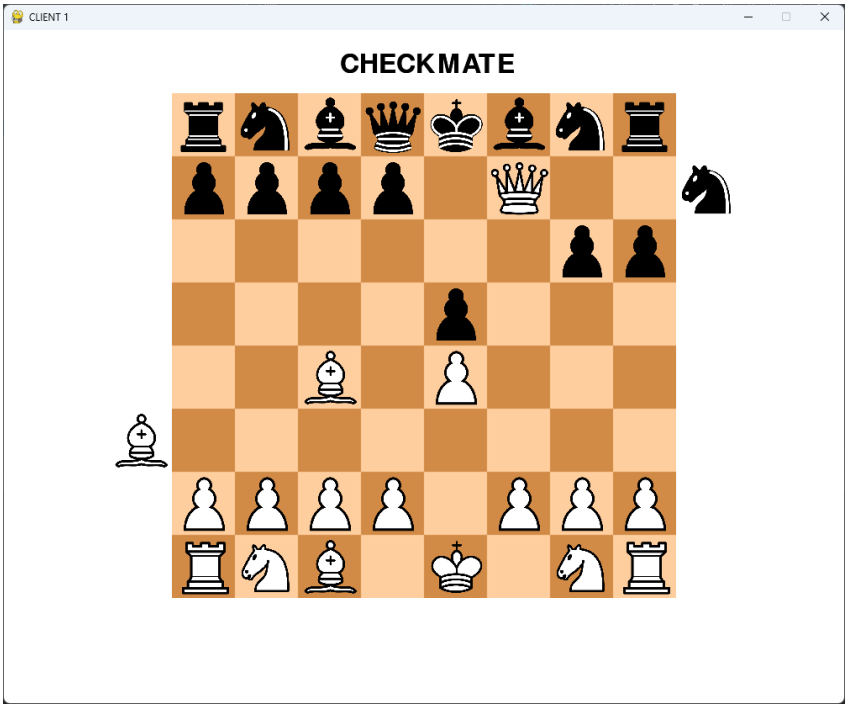
```
    if checkmate:
        while True:
            clock.tick(120)
```

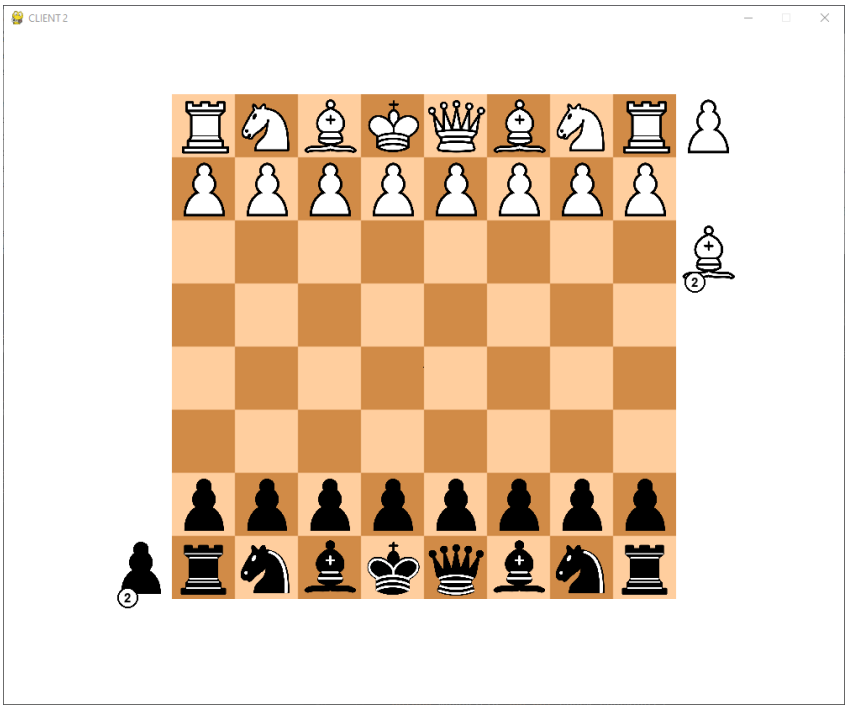
3.2.3 Prototype(s)

Below is a demonstration of the final prototype from Development Cycle 2:

Prototype 2

Here are some images showing the key parts of the new prototype:





3.2.4 Review

Objects Achieved:

- Two connected boards with Two players on each have been implemented
- Captured pieces are sent to the other board
- Captured pieces can be placed by the respective player on the other board
- Checkmate is detected and sent to all boards

At this stage I will present the new prototype to my stakeholder for him to review and use as I plan the stages and aims of my third and final development cycle.

Below is a summary of his feedback:

- The connection between boards was smooth and effective
- He liked the placement of pieces according to their type
- He also liked the bubble to display the amount of pieces on the side of the board
- He reinforced my previous statement that it would be good for more information to be displayed with the checkmate. E.g who checkmated who

- He mentioned that we definitely wants to test it with boards on seperate screens and other players, like an actual game would be played.

This feedback was very helpful to me as a developer, both in reinforcing thoughts I have had throughout the Development Cycle, as well as bringing some other things to my attention. In the next development cycle I will focus on refining some of the details of my prototype (which I will go into more detail on in the next section).

3.3 Development Cycle 3

3.3.1 Aims

In this development I will aim to make some refinements to my solution. This includes things like updating the solution to work across multiple computers, whilst maintaining the same functionality. As well as this it would be good to add clocks to the game as well as more information such as how checkmate has been delivered. If I have time, it would also be good to implement some of the less common chess features, such as **stalemate**, **draw by 50 move rule**, **castling**, **en passant** and **promotion**. However, some of these features are very niche and may require lots of development time and testing which may not be feasible in the given time window. As well as this I would like to make some minor refactorizations to the code, to make sure all the variable names are consistent and clear. This will help with future maintenance and development.

3.3.2 Development

The first thing I would like to do in this development cycle, is update my code so that when checkmate is delivered, the individual boards display more detail about which player has delivered it.

This shouldn't be too difficult due to the way I have coded the server. If the server recognises checkmate whilst processing board 1, then we know that the player who just made a move has delivered checkmate. This is the same for board 2.

Because of this I can simply update the part of the server code that deals with sending checkmate for each board:

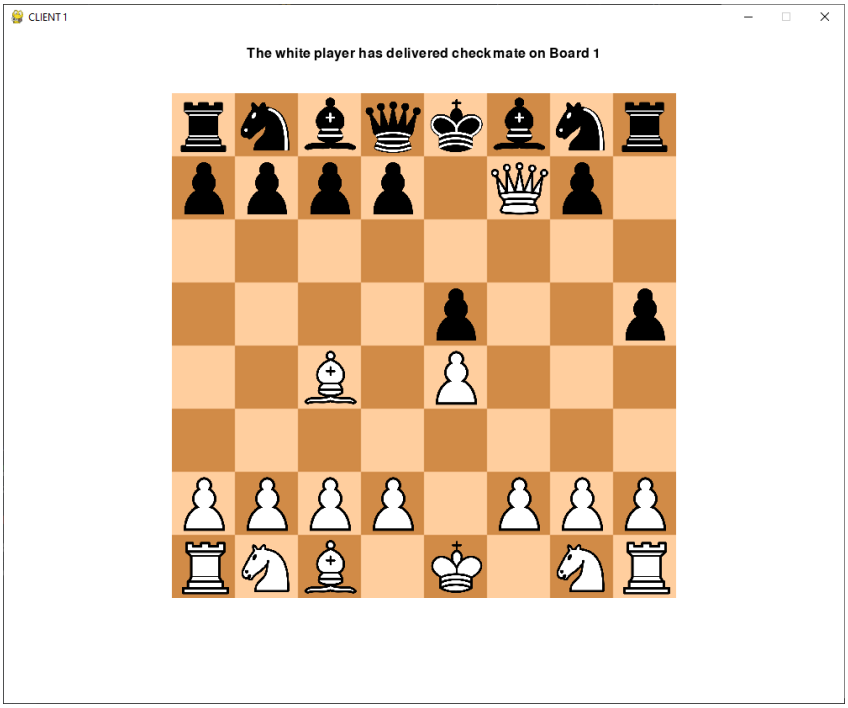
```
    if result == "checkmate":
        for player in client1,client2,client3,client4:
            player.sendall(pickle.dumps(f"checkmate, The {'white'
                                        if request[0][0] == '
                                        w' else 'black'}
                                        player has delivered
                                        checkmate on Board 1"
                                        ))
    else:
        client.sendall(pickle.dumps(result))
```

I think this approach makes most sense. I could send a more concise message to each of the clients representing the same information but this would then require each client to decode it and display the full information, and it is unnecessary to do this 4 times.

I have also formatted it the way I have so that the clients can easily recognise the information. Again I just need to update the original code for each client that handles checkmate to make it deal with the new information:

```
    if type(result) != bool and type(result) != list:
        if result.split(",")[0] == "checkmate":
            #uses the information from the server to directly
                                                display it
            screen.blit(font.render(f"{result.split(',')[1]}", True
                                                , black), (285, 20))
            checkmate = True
```

This shows how a board now displays checkmate information:



The next thing I would like to do is try my code across separate devices, as opposed to running all the clients locally. I don't think I will be able to access 5 separate devices (to act as 4 clients and 1 server), so I will simply run some locally and at least one on a separate device, which should be good enough to show that it will work across multiple devices.

To do this I simply updated the IP addresses and ports in the socket communication. However, I ran into an issue here where the devices would simply not connect. No errors were flagged up but they just attempted to connect indefinitely.

This is likely due to a Firewall or Antivirus on one of the clients, stopping a connection. However, as I am pressed for time I do not have enough time to go down this rabbit hole and find out what is blocking the connection. As far as I can see from my research there is no reason why it should not work.

Although this is unfortunate, I am satisfied that playing the game locally is sufficient and I will be able to make sure the game works across multiple devices in a future development.

3.3.3 Prototype(s)

Below is a demonstration of the final prototype played between four players on one local machine:

Final Prototype

Here is also a demonstration of the checkmate detection, and sending of result:

Final Prototype

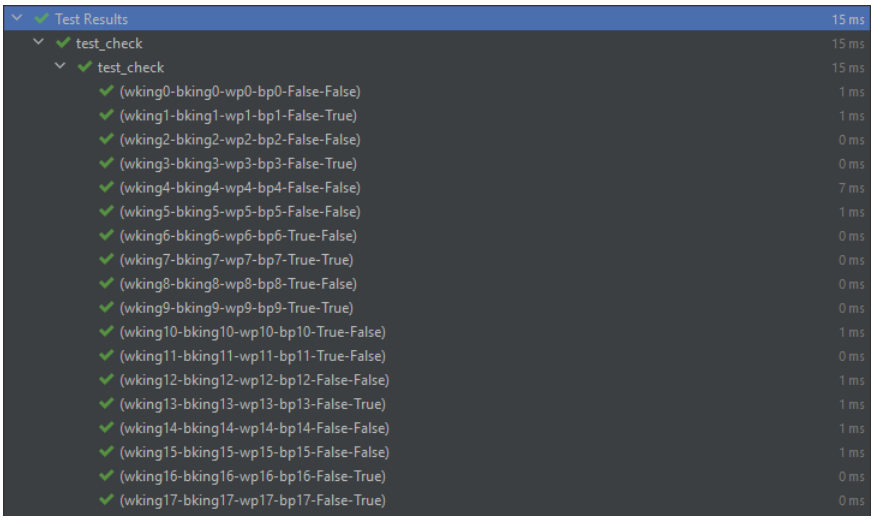
3.3.4 Review

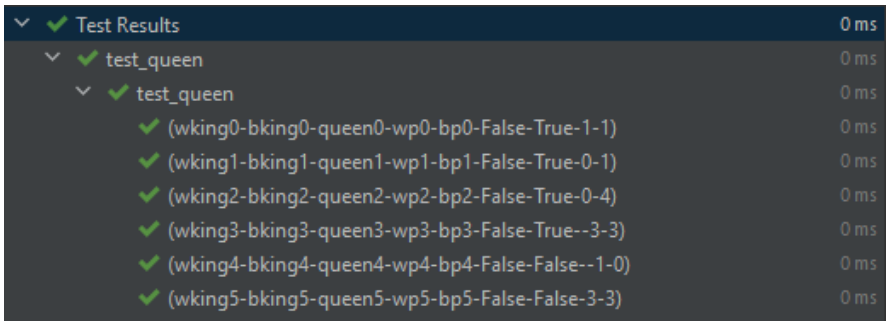
4 Evaluation

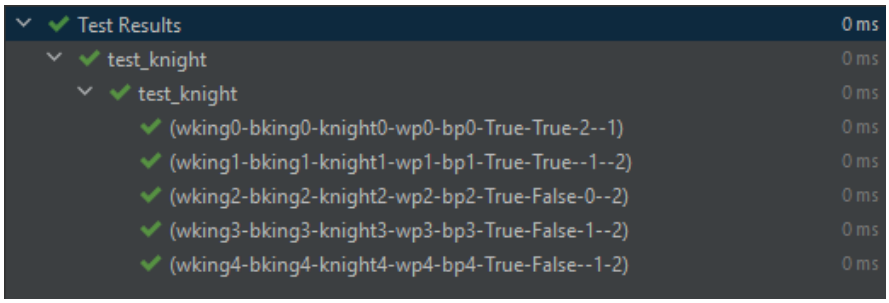
4.1 Testing to Inform Evaluation

4.1.1 Post development testing

Now that I have finished the final project I will run all my unit tests once more, to make sure there are no unexpected failures:







As expected, all the tests still pass which is very reassuring.

I think my tests were quite robust to begin with, and covered many cases. However, with a game with so many possibilities, it is impossible to rest every one. Because of this, what I would implement in the future is a feature where users can report bugs, as in this case, they are the best testers.

4.1.2 Usability testing

My stakeholder was very helpful and managed to find 3 peers to test out the full functionality of the product. He reported lots of success, although there was some tediousness involved with distributing the correct file to each player.

Upon testing, there was success in moving, placing and capturing pieces, with no issues found. As well as this they did not manage to break the product by using invalid inputs which reinforces my earlier statement that the use of a 2D chess board makes it hard to use invalid inputs due to the restriction of physical space. As expected, trying to place a piece off the board is not allowed and the piece simply returns to its original spot. This was handled by the validation performed in the **move_valid()** function, as even though it interprets the attempt as coordinates, as they aren't valid moves the function returns false. The users also reported that they really liked the graphics, everything was clear and a suitable size.

The one issue my stakeholder and his peers ran into was they managed to get their programs to crash by clicking on parts of the screen that were off the board and have no pieces.

4.1.3 Stakeholder Interview

Daniel: Hi Zack. Tell me how you found the product after your testing.

Zack: I found the product really fun to play with and was really impressed by the functionality. The pieces moved really smoothly and were transferred between the boards as expected.

Daniel: Great! I'm really happy to hear that! In our initial interview you mentioned that as a beginner you were hoping for some guidance to be given by your teammate and have the program help you make legal moves. Was that achieved as you were hoping?

Zack: Yeah! The program didn't let me make any illegal moves which was very useful. The help from a more experienced teammate was also very helpful. I did think it may be useful in the future to add sounds to the boards so that is clear when a legal move has been made or when an illegal move has been rejected.

Daniel: Thanks! I completely agree and sounds were something I was intending to do, however I struggled to complete all the more discreet aspects of the project in the time-frame, however there are definitely some features such as sounds and clocks I would like to implement. Speaking of, I know you wanted

the game to stay fast paced, and I was hoping clocks would achieve this. How did you find the lack of clocks.

Zack: Mmm. I was hoping for clocks, however even without clocks the game remained quite fast-paced, so we did not find it too much of an issue. However, I definitely think it would be a good feature to include in the future.

Daniel: Okay, great. Thank you for all your help and enthusiasm during this project, I'm glad I could make a project that fulfilled your requests.

4.2 Success of the solution

User Ratings:

Category	Rating out of 10	Stakeholder Comment
Intuition to use	8	Very intuitive, could use some sounds to help players
Smoothness of game-play	10	Very smooth
Sizing of the UI	7	The size is good but it would be good if the window automatically sizes itself to be full-screen on the device using it
Aesthetics of the UI	7	I really liked the UI, it would be nice if in the future it allowed users to choose different board themes
Fulfillment of essential features	6	Most of the essential features were fulfilled but there is room for improvement by including clocks and sounds

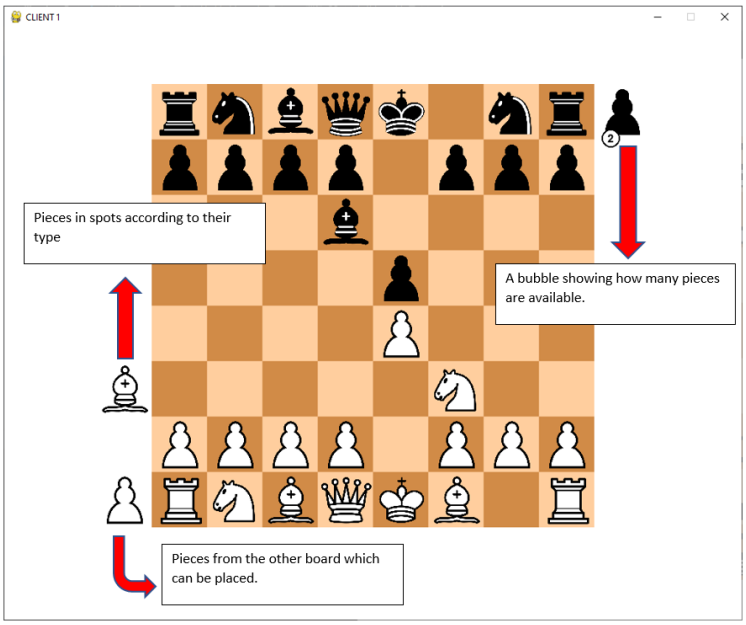
Below I discuss the met/partially met/unmet success criteria, as well as usability features:

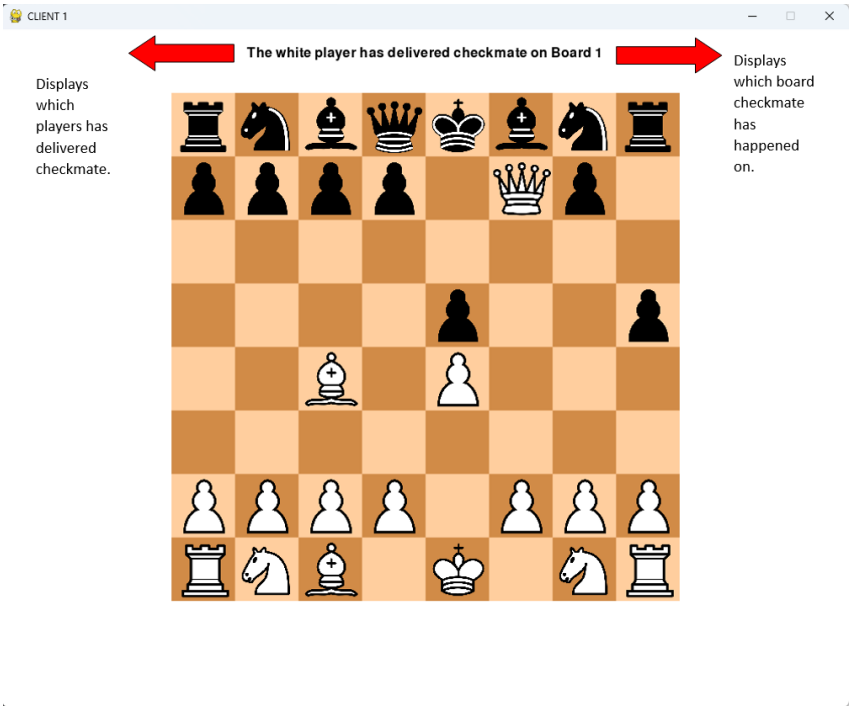
Criteria	Status	Justification
The player should be able to drag and drop pieces to different squares	Met	As shown in the next section, the final product
The program should only allow legal moves to be made	Met	As shown in the unit tests above
Pieces should be able to be captured and placed by a teammate	Met	As shown in the next section, the final product
The program should recognise when a player is in check, and only allow moves that get the player out of check to be made	Met	As shown in the check/checkmate unit tests
The program should recognise when checkmate has been delivered	Met	As shown in the checkmate unit test
The colours of the pieces and board should be clear	Met	As seen in user feedback
The pieces should be big enough and each type of piece should be clearly recognisable	Met	As seen in user feedback
The program should allow special moves such as castling, promotion and en passant to be made	Unmet	I deemed these features too niche to spend time on, given the limited timeframe to create the solution
The program should recognise when a draw has been made in the form of stalemate, repetition or the 50-move rule	Partially met	I simply did not have time to add in these features. The 50-move rule should be quite simple to implement but draw by repetition would require storing all previous states of the game which would require lots of memory and may be unfeasible. Stalemate would probably in the middle of these two in terms of difficulty, however its not extremely essential as if there is a stalemate the program will not allow players to make a move, so the functionality is partially implemented.

It can be seen from above that I have had quite a lot of success with my solution. That being said, it was clearly a project with lots of content from the beginning, not all of which I have been able to complete. However, lots of this content is stuff that I will be able to develop and improve in the future.

4.3 The Final Product

Here are some annotated images, showing the features of the interactive chessboard:





Most of the features involve playing the game and moving the pieces, so here is a video demonstrating that: Final Prototype

Here is a video demonstrating the checkmate detection, with updated details given:

Final Prototype - Checkmate

Here is the GitHub repository with all the relevant files, as well as details on how to run them in the README:

https://github.com/Daniel-Groves/Daniel-Groves—Bughouse

4.4 Maintenance and Further Development

4.4.1 Maintenance

Due to the way I have coded my solution it should be quite easy to maintain my project. My project is heavily modular which should make it easy for myself or other developers to understand and maintain in the future. As well as this it is well commented which should aid understanding in the future.

Due to the object oriented approach it would also be easy in the future to change the starting pieces and play **Fischer Random chess**.

One thing that I could have done better on this front would have been to "draw" the board in pygame, which would then make it a lot easier to change the colour/style of it in the future. The way I did it with an image, makes this hard to do.

4.4.2 Further Development

There are quite a lot of further developments I can make to my project. Both ones that I initially planned to accomplish in this project, as well as ones that I did not plan to do.

- Castling, promotion and En Passant - These could be included with some development of the code to allow for these features, and would not require a significant rewrite
- Recognise multiple game endings (Stalemate, draw by repetition, draw by 50 move rule). As above, this would require some additional code, but not a major rewrite
- Clocks. This would require some additional development on the server and client side to maintain synced clocks between all the players, but could significantly improve game play.
- Sounds. I don't expect this would be too difficult to implement but would require some additional research in playing sounds in python.
- More customisation options. I would like to provide more customisation options in the future, e.g board style and color however this would require a significant amount of additional code. I would need to add an interface for users to select their preferred options and also add functionality to change the board.
- Allow players to play across multiple separate devices. This was something I originally planned to do towards the end of my project, however unforeseen issues with socket and timing made it unfeasible. It is definitely one of the first things I would implement in future development.